Universidade do Minho
Escola de Engenharia

Marta Susana Gonçalves Pereira

Motor de Eventos

Setembro de 2009

Marta Susana Gonçalves Pereira **Motor de Eventos**

UMinho | 2009

Universidade do Minho
Escola de Engenharia

Marta Susana Gonçalves Pereira

Motor de Eventos

Tese de Mestrado em Computação Gráfica e
Ambientes Virtuais

Trabalho efectuado sob a orientação do
Professor Doutor António Ramires Fernandes

Setembro de 2009

Anexo 3

DECLARAÇÃO

Nome Marta Susana Gonçalves Pereira

Endereço electrónico: msgpereira@gmail.com Telefone: 965390387

Número do Bilhete de Identidade: 10929443

Título dissertação ?/tese ?

Motor de Eventos

Orientador(es): Professor Doutor António Ramires Fernandes

Ano de conclusão: 2009

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Computação Gráfica e Ambientes Virtuais

Área de Especialização em Informática

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, ___/___/_____

Assinatura: _____

AGRADECIMENTOS

Agradeço ao Professor Doutor António Ramires Fernandes, pela competência com que me orientou durante a dissertação e pela confiança que sempre me incutiu.

Aos meus três meninos, agradeço pelo incentivo constante, pela paciência e pelas vezes que não lhes consegui dar mais e melhor atenção.

Aos amigos que directa ou indirectamente contribuíram para a elaboração deste projecto.

RESUMO

"A Universidade do Minho, em particular o grupo de Gráficos, Interacção e Visão (GIV), em parceria com o Município de Ponte de Lima, desenvolveu um trabalho no âmbito do projecto 'Ponte de Lima, Terra Rica da Humanidade', que consistiu na modelação 3D, e apresentação sobre diversos formatos, da vila de Ponte de Lima na sua versão actual no início do século XXI, e numa possível reconstrução virtual da muralha construída no século XIV. De um ponto de vista académico e no âmbito da computação gráfica, este projecto permitiu ao grupo GIV atingir dois objectivos: a construção de um cenário virtual de grandes dimensões e um motor de renderização 3D completo e extremamente versátil." [pl3D]

A transcrição anterior refere-se ao motor gráfico de renderização 3D denominado Curitiba. Este permite visualizar modelos 3D, com capacidade para realizar múltiplos passos, possibilitando a geração de efeitos visuais complexos. Embora a demonstração do motor, no âmbito do projecto Ponte de Lima 3D, permitisse algumas animações e interactividade, estas foram de facto criadas através da alteração do próprio código, sem que houvesse na altura muito cuidado na elaboração de uma arquitectura para criar cenas dinâmicas. A interacção com alguns elementos no modelo, como por exemplo, clicando na marca que indica a subida do rio numa das grandes cheias ocorridas, provoca uma animação. Essa animação foi realizada de forma muito direccionada para o efeito desejado nesse momento, não permitindo generalizar de forma sistemática a criação de outras situações com animações diferentes, ou, mantendo animação, aplicá-la a outros modelos.

Apesar de ter acoplado um motor de física e estar dotado de capacidades para facilmente ser expansível, o Curitiba permitia apenas criar um ambiente 3D estático sem interacção, a menos da movimentação da câmara no modo de primeira pessoa.

Neste contexto pretende-se dotar o Curitiba da capacidade para definir e visualizar cenas dinâmicas, e com interacção, quer com o utilizador quer entre elementos da cena. É nesta perspectiva, a de acrescentar funcionalidade e interacção ao ambiente, que é proposto realizar nesta dissertação de mestrado, um estudo e implementação de uma arquitectura de software para um motor de eventos. Um motor de eventos que permita acrescentar diferentes tipos de interacção num ambiente 3D, que seja fácil de usar, genérico, e expansível, seguindo a própria filosofia do Curitiba.

ABSTRACT

"Minho University, in particular the Graphics, Interaction and Vision team (GIV), in association with the council of Ponte de Lima has developed a project called 'Ponte de Lima, a Land Rich in Humanity', that consisted of the 3D modeling and presentation, in various formats, of the town of Ponte de Lima in its present version of the beginning of the XXIst century and in a virtual prospective reconstruction with the wall built in the XIVth century. From an academic point of view, and in the graphic computation field, this project allowed the GIV team to achieve two purposes: the construction of a large virtual scenario and a complete 3D rendering engine that is extremely versatile." [pl3D]

The previous quotation refers to a 3D rendering graphic engine called Curitiba. This allows 3D model visualization with multiple steps which make complex visual effects possible. Although the engine demonstration in the "Ponte de Lima 3D" project allowed some animation and interactivity, in fact they were achieved with the change of the engine's own code, without taking into consideration at the time, the building of a dynamic architecture to create dynamic scenes. The interaction of some elements in the model, for instance, by clicking on a mark which represents the flooding of the river, causes an animation. That animation was achieved based on the wanted effect at the time, but it can't be applied systematically when creating other situations with different animations or, when using that very animation on other models.

Despite having engaged a physics engine and being able to be easily expandable, Curitiba, at the present moment, creates a static 3D ambience without interaction, besides the movement of the camera in first person mode.

The aim of this work is to provide Curitiba with the ability to define and visualize dynamic scenes and with interaction among both the user and elements of the scene. To accomplish this goal, to add functionality and interaction to 3D environment, we propose to accomplish in this master work a software architecture study and implementation for an event engine. An event engine that permits expansion of different interaction types in 3D environment that is easy to use, generic and expandable, following the very philosophy of Curitiba.

ÍNDICE GERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Capítulo 1 - INTRODUÇÃO

Entende-se por motor como um conjunto de blocos de software pré-fabricados que podem ser usados como base de desenvolvimento para novas aplicações. Os motores prevêem um comportamento padrão para a maioria das características e funcionalidades existentes.

Um Motor de jogo (*game engine*) é um programa de computador e/ou conjunto de bibliotecas, para simplificar e abstrair o desenvolvimento de jogos ou outras aplicações com gráficos em tempo real. [MotorDeJogo]. Um motor de jogos tem duas componentes muito importantes, que são: o motor gráfico e o motor de física.

Os motores gráficos são responsáveis por processar descrições gráficas dos modelos gerar primitivas gráficas de baixo nível. Como exemplo, pode-se citar: *Crystal Space, Irrlicht, OGRE*, e *RealmForge*.

Os motores gráficos incluem renderização, a movimentação da câmara, e efeitos gráficos definidos pelo utilizador para, por exemplo, criar uma visualização realista. Podem também ser responsáveis pela animação de objectos.

Os motores de física são responsáveis por simular acções reais, aplicando as leis da física recorrendo a definições que incluem a gravidade, massa, fricção, força, flexibilidade, etc... Como exemplo, pode-se citar: *Bullet,* e *ODE* [MotorDeFísica].

Através dos motores de física é possível realizar a movimentação da câmara impedindo que esta intersecte a geometria do mundo virtual e garantindo que a câmara segue o terreno, quer determinando colisões entre uma geometria virtual que represente um avatar ou veículo, quer aplicando a lei da gravidade.

Um outro componente necessário a um motor de jogos é o motor de eventos. Entende-se como evento uma colecção de dados produzidos por um componente do sistema e que são do interesse de zero ou mais componentes do sistema. [Gross].

Desta forma, algum componente do sistema, o motor gráfico por exemplo, pode produzir um evento que será do interesse de um ou mais componentes. A título de exemplo temos o motor gráfico que sempre que inicia ou termina uma frame, produz um evento com um selo temporal que poderá ser utilizado para controlar animações.

Seguindo esta linha de visão, podemos ainda considerar que um evento pode desencadear vários eventos, que por sua vez desencadearão outros ainda. Por exemplo, a câmara aproxima-se de uma porta que se abre e que por sua vez acende a luz, que acorda o gato que sai disparado a correr porque ficou assustado.

A gestão de eventos deve ser simples, eficiente e genérica para que permita a maior interacção possível num ambiente, dando liberdade ao utilizador para criar animações e/ou definir interacções com um mínimo de restrições e complexidade.

Desta forma, a gestão de eventos pode e deve ser realizada por um motor específico para o efeito, isto é, um motor de eventos.

Esta tese tem como objectivo a criação de um motor de eventos simples, genérico e facilmente expansível.

Pretende-se elaborar uma arquitectura para a criação de cenas dinâmicas, acrescentando ao Curitiba, acima de tudo, interacção quer com o utilizador quer entre elementos da cena.

O Curitiba possuirá, desta forma, a capacidade para definir e visualizar cenas dinâmicas.

A filosofia de implementação do Curitiba tem como principal pilar a expansibilidade. A arquitectura do Curitiba está desenhada de tal modo que a implementação de novas funcionalidade exige um mínimo de reescrita do próprio Curitiba. O motor de eventos pode, e deve, ser construído com um nível mínimo de intrusão no código original.

É também um requisito deste projecto que o fluir dos eventos possa ser especificado nos ficheiros de projecto que permitem definir cenas, até agora estáticas, com recurso à notação XML.

1.1. Motivação e Objectivos

O Curitiba é um motor gráfico, que permite visualizar modelos 3D, com capacidade para realizar múltiplos passos, permitindo a geração de efeitos visuais complexos.

A principal motivação desta tese foi a criação de um mecanismo que permitisse acrescentar interacção ao Curitiba. Interacção, como já foi referida, que pode ser entre o utilizador e o motor gráfico, entre o utilizador e os elementos da cena, ou entre os próprios elementos da cena.

Pretende-se que este mecanismo seja simples na sua utilização por forma a motivar os utilizadores do Curitiba a tirarem partido do mesmo. Seguindo a filosofia de desenvolvimento do próprio Curitiba, o principal objectivo não é a definição de todas as formas de interacção possíveis, mas antes a estruturação e implementação de uma arquitectura que não limite a expansibilidade e inclua os mecanismos essenciais para a definição de futuras formas de interacção e animação. Tal implica que o núcleo do

sistema de animação seja genérico, e pensado de forma a que necessite de um mínimo, ou idealmente de nenhuma, intervenção para adicionar novas formas de interacção/animação.

Após uma análise a vários motores de jogos, verificou-se que o mecanismo pretendido pode ser obtido com um motor de eventos.

Se juntarmos ao Curitiba um motor de eventos, simples, genérico e expansível, passamos a obter uma ferramenta poderosa e com um grau de flexibilidade muito superior ao existente no início deste trabalho.

O objectivo não é criar um motor de eventos que permita definir animações complexas, mas sim estruturar e implementar um motor que crie o fluxo de eventos que despoleta e controle essas mesmas animações.

O motor deverá ainda permitir formas de interacção com o motor gráfico que facilitem a sua utilização num ambiente de investigação, ou seja, permitir que o motor gráfico seja configurável em tempo real, por exemplo enviando mensagens aos objectos para que utilizem um determinado material, ou alternar entre técnicas de visualização.

Por construção, o Curitiba foi estruturado por forma a minimizar as intervenções no seu núcleo, quando se pretende adicionar novas funcionalidades. O motor de eventos deverá respeitar a filosofia actual do motor gráfico, e portanto funcionar de forma autónoma, com um mínimo de invasão no núcleo do Curitiba.

A definição de eventos, e o seu fluxo, deverão ser facilmente configuráveis pelo utilizador, quer através de programação de uma aplicação que utilize o Curitiba, quer através da especificação dos mesmos nos ficheiros de configuração XML.

A utilização dos ficheiros de configuração permitirá usufruir do mecanismo de fluxo de eventos, sendo apenas necessário estender a API do Curitiba para suportar os tipos de eventos requeridos.

1.2. Estrutura da Tese

No capítulo seguinte, capítulo 2, será apresentado inicialmente um breve historial sobre a evolução dos jogos seguido de um estudo sobre Motores relacionados e as suas componentes principais, em particular os Motores de Eventos, nomeadamente nos motores de jogos Irrlicht e Ogre.

Será ainda aprofundado o processamento de eventos no VRML, uma vez que o modelo do motor de eventos do VRML contribuiu largamente para o desenvolvimento da arquitectura que será apresentada no capítulo 3.

O capítulo 3, reservado para o desenvolvimento, apresenta a arquitectura do motor de eventos, a gestão e processamento de eventos. É abordado o processo de reencaminhamento e transformação dos eventos recorrendo a uma estratégia semelhante à utilizada no VRML. São também, efectuados testes ao motor de eventos, com exemplos por ordem crescente de grau de dificuldade, apresentando sempre soluções alternativas e construtivas no sentido de melhorar e aperfeiçoar a implementação do motor de eventos. No final do capítulo, é feita uma apresentação de como definir cenas dinâmicas carregando ficheiros de configurações em XML.

Finalmente, no último capítulo, é feita uma reflexão crítica sobre o trabalho desenvolvido e são apresentadas algumas sugestões de trabalho futuro.

Capítulo 2 - ESTADO DA ARTE

Desde os tempos antigos que o homem joga para desenvolver-se física e intelectualmente, mas também pelo seu instinto de competição. A Era Tecnológica evoluiu os jogos, criando ambientes complexos tanto para reprodução visual quanto para resolução do enredo.

O primeiro jogo de computador foi o Pong [Pong], que basicamente movimentava uma bola de um lado para o outro como num jogo de ténis.

Com o evoluir da tecnologia, mais especificamente os computadores, começaram a surgir maior variedade de jogos de computador, cada vez mais avançados e apelativos, com vários níveis de dificuldade, despertando cada vez mais o interesse do utilizador.

Até cerca dos anos 90 os jogos eram apenas 2D, passando nessa década ao conceito 3D. O primeiro jogo 3D foi o Wolfenstein 3D (Figura 1), da empresa Id Software em 1992. Era um jogo do tipo *"First-person shooter"* [Deligiannidis], os itens e os inimigos eram *sprites* (impostores - uma colecção de imagens estáticas) que se moviam pelo ambiente, aumentavam e diminuíam o tamanho, e não possuíam animação. Não permitia saltar ou subir escadas. Depois deste jogo, a empresa *Id Software* produziu o e mais tarde o Doom (Figura 2) e de seguida o Quake (Figura 3) que já adicionou itens e inimigos em 3D.

Figura 1. Wolfenstein 3D

Figura 2. Doom

Figura 3. Quake 3

Os jogos de computadores modernos fazem uso de tecnologias de várias áreas da computação, tais como gráficos, inteligência artificial, programação em rede, sistema operativos, linguagens e algoritmos. Estes contêm uma grande variedade de implementações eficientes de técnicas de renderização e interacção. Geralmente, concentram-se na visualização de um ambiente 3D e a interacção com os elementos visíveis. [Scoresby]

2.1. Motores de Jogos

Um ambiente gráfico 3D é a simulação de um espaço tridimensional onde podemos observar os objectos nele contidos de vários ângulos e distâncias, navegando pelo ambiente, mudando de posição e rodando nesse espaço.

A produção de jogos 3D de alta qualidade normalmente envolve um grupo elevado de pessoas que despendem muito tempo na sua implementação. Um projecto desta dimensão implica, para além de tempo, um orçamento elevado, uma vez que são necessários profissionais qualificados de diversas áreas: designers; programadores, animadores, artistas gráficos, artistas sonoros, etc. A reutilização de software torna-se imperativa como método de controlo de custos. Surge desta forma o conceito de motor de jogos (*game engine*). [Haque] [Hannah]

O termo "*Motor de Jogos*" surgiu na década de 1990, interligado com os jogos 3D (*FPS - First Person Shooter*), tais como, o Doom e o Quake. A ideia subjacente é que os motores implementem funcionalidades e recursos comuns à maioria dos jogos de determinado tipo, permitindo que esses recursos sejam reutilizados a cada novo jogo criado.

A programação de sistemas desenvolvidos com técnicas de programação orientada a objectos veio permitir o melhoramento dos jogos de computadores, uma vez que é possível acrescentar novas funcionalidades e tirar proveito das já existentes sem perder a eficiência e jogabilidade. A versatilidade e a reutilização de código são, sem dúvida, a chave para o acompanhamento da evolução dos jogos que aumenta a grande velocidade.

A tecnologia dos motores de jogos hoje em dia permite um eficiente e efectivo desenvolvimento do mundo virtual. Podem ser usados para criar mundos interactivos 3D que se baseiam na qualidade do fotorealismo dos gráficos e o alto nível das navegações imersivas. [Andreoli] [Jiang]

Normalmente as funcionalidades de um motor de jogos são: Rendering; Áudio; Animação; Inteligência Artificial; Comunicação em Rede; Scripting; Colisão/Motor Físico. Estes sistemas podem ser implementados de forma independente, sem que interfiram uns com os outros. A seguinte figura (Figura 4) mostra o desenho genérico, ao mais alto nível, de um motor de jogos. [Wünsche] [Janc]

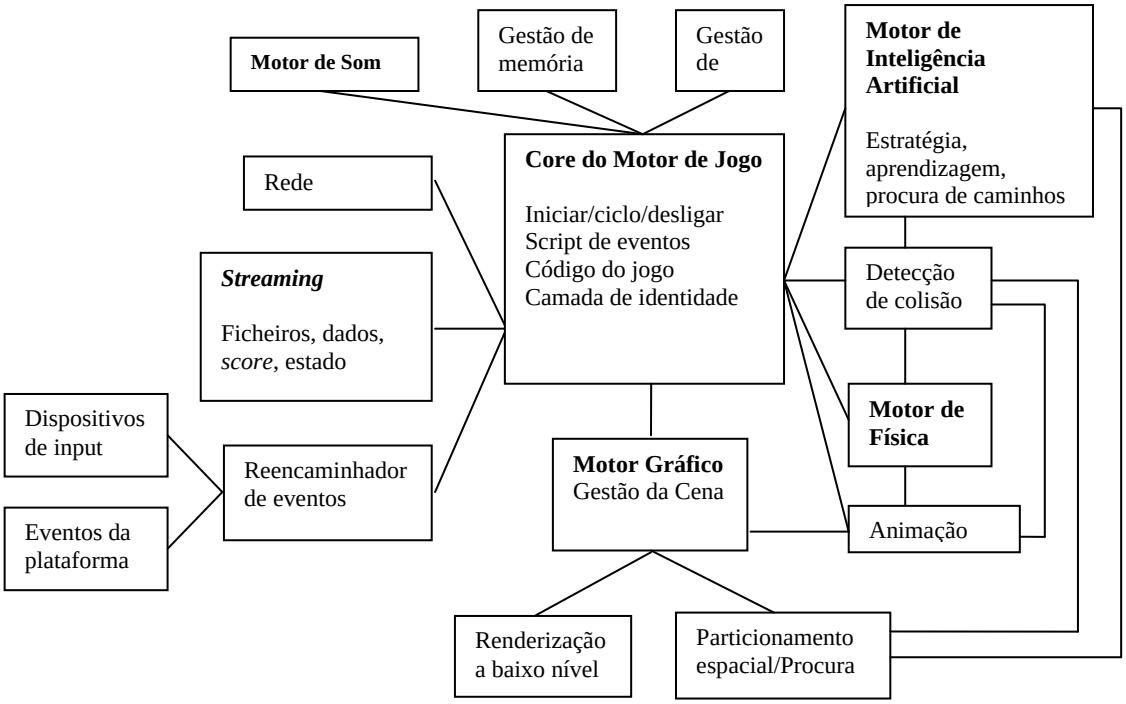

Figura 4. Arquitectura de um Motor de Jogos ao mais alto nível

Motor Gráfico (Graphics Engine)

A sua funcionalidade é fazer o render dos objectos 3D. É normalmente baseado em APIs de baixo nível para DirectX e OpenGL.

Geralmente suportam APIs de médio nível e de alto nível, que proporciona portabilidade e grafo de cena, respectivamente. O grafo de cena, é uma estrutura de dados que tem toda a informação necessária para fazer render da cena e tem uma interface simples de usar para o utilizador.

Motor Físico (Physics Engine)

Proporciona movimentos realísticos e efeitos da interacção entre os objectos do ambiente

Os componentes do motor físico são: Colisão (detecção e resposta); Fluidos; Partículas; Corpo Rígido; Junções; etc. [Bongart] [ODE]

Exemplos de Motores Físicos:

- *Havok Engine* (www.havok.com)
- *RenderWare Physics Engine* (www.renderware.com)

Motor de Inteligência Artificial (AI Engine)

Virtualmente todas as técnicas usadas em inteligência artificial, são úteis num jogo. Inclui uma máquina de estados finita, aprendizagem, procura de caminhos, cooperação de agentes múltiplos, redes neuronais, algoritmos genéticos, processamento de linguagem natural, etc. [Li] [Lester]

Exemplos de Motores de IA:

- *AI.implant* (www.ai-implant.com)
- *Spark* (www.louderthanabomb.com)
- *PathEngine* (www.pathengine.com)
- *RenderWare AI Engine* (www.renderware.com)

Motor de Áudio (Sound Engine)

Os motores de áudio suportam áudio digital 2D, áudio digital 3D, entrada e saída de áudio digital, formato de áudio comprimido (MP3, OGG, etc.), e tecnologia de plug-in 3D (EAX, Dolby, RSX, etc.)

Exemplos de Motores de Áudio e plug-ins

- *The Miles Sound System* (www.radgametools.com)
- *Creative* (developer.creative.com)
- *Thomson* (mp3licensing.com)
- *Ogg Vorbis* (www.vorbis.com)
- *Dolby* (www.dolby.com)

Existem vários motores de jogos, alguns freeware, outros comercializáveis, como por exemplo:

- *OGRE* (www.ogre3d.org)
- *Irrlicht* (irrlicht.sourceforge.net)
- *Crystal Space* (www.crystalspace3d.org)
- *Torque Game Engine* (www.garagegames.com)
- *3DgameStudio* (www.3dgamestudio.com)
- *Panda3D* (*www.panda3d.org*)

A maioria dos motores de jogo são distribuídos em forma de API, sendo alguns motores distribuídos com um conjunto de ferramentas, que agiliza e simplifica ainda mais o desenvolvimento do jogo, como por exemplo: scripts pré-programados, e

programas que “empacotam” e deixam pronto o jogo para distribuição. Esses últimos motores referidos são chamados de *Middleware*. Pelo facto de serem distribuídos com essas funcionalidades, eles suprem a necessidade da compra de outras ferramentas, reduzindo os custos. Como exemplo temos o *Blender, Havok, RAD Game Tools, RenderWare* e, *Unreal Engine*. [MotorDeJogo] [Tulip].

Alguns motores de jogos usualmente fornecem abstracção de hardware, permitindo que um mesmo jogo, seja executado em diferentes plataformas. A maioria dos motores de jogos multiplataforma, são desenvolvidos usando APIs, como o *OpenGL*, *DirectX*, *OpenAL* e *SDL*. [MotorDeJogo]

2.2. Motores de Eventos

Esta dissertação de mestrado, tal como foi mencionado anteriormente, centra-se na criação de uma arquitectura de um motor de eventos que seja suficientemente abrangente para permitir acrescentar vários tipos de eventos de forma fácil e independente. Pretende-se com esta gestão de eventos, acrescentar interacção num ambiente 3D quer com o utilizador quer entre elementos da cena.

Normalmente a interacção com o ambiente é desencadeada por dispositivos de input, ou colisão de objectos. Desta forma, quando um evento deste tipo é recepcionado pelo motor de eventos terá de ser reconhecido e, por exemplo, reencaminhado para o motor responsável por desencadear a consequência desse evento, por exemplo, motor de física, Inteligência Artificial, Áudio, Gráfico, etc. Basicamente o motor de eventos irá funcionar como intercomunicador entre os restantes motores, ou directamente com os objectos.

É importante considerar que comportamentos poderão ser produzidos, para prever que informação poderá ser necessária inferir. Se pretendemos permitir a possibilidade de criar efeitos usando física, teremos de interpretar informação importante, tal como, a velocidade de impacto numa colisão, para que essa informação em conjunto com as propriedades do material permita simular fenómenos físicos, como quebrar um copo, ou janela, etc. [Lugrin]

Na maioria dos casos, os eventos derivam de mecanismos de detecção de colisão pelo motor gráfico. Estes sistemas são capazes de produzir primitivas de eventos que correspondem a colisão ou contacto entre objectos, assim como, objectos a saírem ou entrarem em determinadas áreas ou volumes num ambiente 3D.

Existem alguns artigos que descrevem a funcionalidade e implementação de alguns sistemas, desde sistemas de realidade virtual, jogos, etc., e que fazem referência à forma genérica como interpretam os eventos. A maioria usa *Listeners* (ouvintes) que são classes que permitem receber notificações de eventos.

O motor de jogo *Unreal Tournament* trata os eventos enviando mensagens para actualizar o estado dos objectos que depois são interpretados [Cavazza].

Algumas aplicações com GUI para ambientes virtuais funcionam usando um filtro de eventos que pode parar ou reencaminhar um evento para outros objectos [Andujar].

Os sistemas inteligentes baseados em modelos de eventos conhecidos como *Active Events* (ou *ActEvents*) tratam da comunicação entre os dados de origem e as acções que estes dados provocam, normalmente aplicado em sistemas distribuídos, usam para comunicar mensagens estruturadas do tipo XML [Gross].

No desenvolvimento de jogos para dispositivos móveis, também os listeners são muito utilizados. Por exemplo, no motor de jogos **M3GE,** a classe *UpdateListener* deve ser implementada pelo programador do jogo caso haja necessidade de tratar alguma acção nas rotinas de renderização, ou alguma lógica de jogo a ser adicionada em termos de eventos. Isto permite, por exemplo, que o programador de jogos disponibilize informações detalhadas para os jogadores, escrevendo-as directamente no ecrã. Os principais eventos disponíveis são: *camUpdate*, gerado quando o jogador acciona a troca de câmara; *fire*, gerado quando o jogador atira e acerta nalgum objecto 3D; *keyPressed* e *keyReleased*, gerados quando o utilizador pressiona ou solta, respectivamente, alguma tecla do dispositivo; *update*, gerado antes de ser redesenhada a cena; e *paint* gerado após redesenhar a cena. Este último evento permite, por exemplo, que seja desenhado algum elemento 2D no dispositivo gráfico, após o desenho da cena 3D. [Gomes] [Gustavsson]

Na simulação de eventos físicos numa mesa interactiva, ao nível de abstracção mais alto, a aplicação de gestão de eventos permite que o objecto notifique eventos predefinidos, que o sistema irá devolver a outros objectos que estão a escuta ou que esperam por eles. Normalmente, este tipo de gestão de eventos é implementado usando observadores de padrões, onde os objectos se registam como observadores de uma classe de eventos que desejam. [Moreno]

No processamento de eventos em VRML, um evento significa informação que deve ser transferida entre dois nós, no grafo de cena. Por exemplo, os inputs gerados pelo utilizador são capazes de produzir uma fonte de animação de alguns nós no grafo de

cena. Uma vez gerados esses eventos, são enviados para o destino traçado, por ordem de tempo, e processados pelo nó que os recebe. Este processamento pode mudar o estado de um nó, gerar eventos adicionais ou mudando a estrutura do grafo de cena. Este importante processo é denominado de reencaminhamento de eventos (*event routing*). Portanto, a mudança de um ambiente virtual ou grafo de cena é causado por estas interacções e animações. [Kim]

Este conceito no processamento é interessante e com grandes capacidades, pelo que será feita uma análise mais detalhada do processamento de eventos em VRML.

2.3. Processamento de Eventos em VRML

No VRML o grafo de cena é construído por grupos de objectos dispostos hierarquicamente. Um nó é um elemento do grafo que implementa uma determinada funcionalidade. Estes nós agrupam-se numa hierarquia criando uma estrutura definida de pai para filhos.

Cada nó tem um certo número de campos que expressam os seus atributos e as suas propriedades. Os campos que podem receber eventos são denominados de *eventIns*. Os eventos que um nó pode gerar estão associados a campos do tipo *eventOut*. Quando um campo pode ser o receptor dos dados de um evento, e por sua vez também é utilizado no envio de eventos diz-se que é um *ExposedField*. O fluxo de eventos é definido através de *routes*. Um *route* estabelece uma conexão entre os *eventOuts* e os *eventIns*, sendo a compatibilidade dos tipos de dados associados ao *eventOut* e *eventIn* a única restrição imposta [Taniguchi] [lighthouse3D].

Inicialmente um evento é enviado, seja por acção interactiva do utilizador com um objecto, ou por um temporizador, por exemplo, e é reencaminhado para os nós de destino por acção dos routes. Por sua vez quando esses nós recebem o evento podem gerar outros eventos e assim sucessivamente.

Este fluir de eventos cria um conceito de interacção bastante abrangente.

O route define-se com a seguinte sintaxe:

```
ROUTE Node.eventOut_changed TO Node.set_eventIn
```

Exemplo: Suponhamos que definimos um *PlaneSensor*, que é um sensor que permite deslocar objectos nesse plano, e uma esfera. Pretendemos mover a esfera nesse

plano. Teremos de criar o *PlaneSensor*, a esfera e o route. Sempre que o rato se movimenta no plano envia um evento com a posição do rato. O route direcciona o evento gerado pelo *PlaneSensor* para a esfera. Desta forma, podemos fazer a interligação entre eles usando o route, definindo que ao mudar a posição do rato no plano, a esfera deverá assumir essa nova posição como sua.

```
#VRML V2.0 utf8
Group {
    children [
        DEF ts PlaneSensor {
            minPosition -1 -1
            maxPosition 1 1
        }
        DEF tr Transform {
            children Shape {geometry Sphere {}}
        }
    ]
}
ROUTE ts.translation_changed TO tr.set_translation
```

2.4. Processamento de Eventos no Irrlicht

O IrrLicht é um motor gráfico multiplataforma, cujo código é disponibilizado sem restrições. Tendo como base uma pequena equipa de desenvolvimento, e tem tido grande aceitação por parte da comunidade. As funcionalidades mais importantes do motor estão descritas na tabela 1.

Características gerais	O motor é completamente orientado a objectos e independente da plataforma Possui um mecanismo para carregar cenas Possui um interpretador de XML
Editores	irrEdit – editor de cena e construtores de mapas de luz Editor GUI
Física	Detecta colisão (Bounding box e triangle based collision) e resposta à colisão
Iluminação	Por pixel, por vértice, mapeamento

Sombras	Volumes de sombras
Texturas	Aplicação de texturas, Multi-texturing, Bump mapping, Mipmapping
Shaders	Vertex, Pixel
Gestão da Cena	Gestão geral, BSP, Octree
Animação	Animação de esqueleto, Morphing, Animation Blending
meshes	Carregamento de vários tipos de meshes
Efeitos especiais	Environment Mapping, Billboarding, Particle System, Sky, Water, Fog
Terreno	Rendering, CLOD
Rendering	Fixed-function, Render-to-Texture, Fonts, GUI

Tabela 1. Funcionalidades mais importantes do motor Irrlicht

O processamento de eventos no Irrlicht é feito pela notificação de eventos para uma cadeia de receptores de eventos. Não existe um motor específico para controlar o fluxo de eventos, simplesmente são enviados de uns objectos para os outros. Desta forma um evento não tem um destinatário específico, nem é possível definir um.

Quando ocorre um evento este é passado por uma cadeia de receptores. Os receptores podem ser a janela, o ambiente GUI e os seus elementos, o receptor de inputs do gestor de cena, a consola, a câmara activa e os receptores do utilizador. Estes últimos, receptores do utilizador são classes definidas pelo utilizador como receptoras de eventos. A ordem pela qual passam pela cadeia de receptores depende do tipo de evento ocorrido, como pode-se constatar na Figura 5. Por exemplo, se um evento de tecla ocorre este é passado aos receptores do utilizador, depois ao ambiente GUI e os seus elementos, de seguida ao receptor de inputs do gestor de cena e por fim à câmara activa.

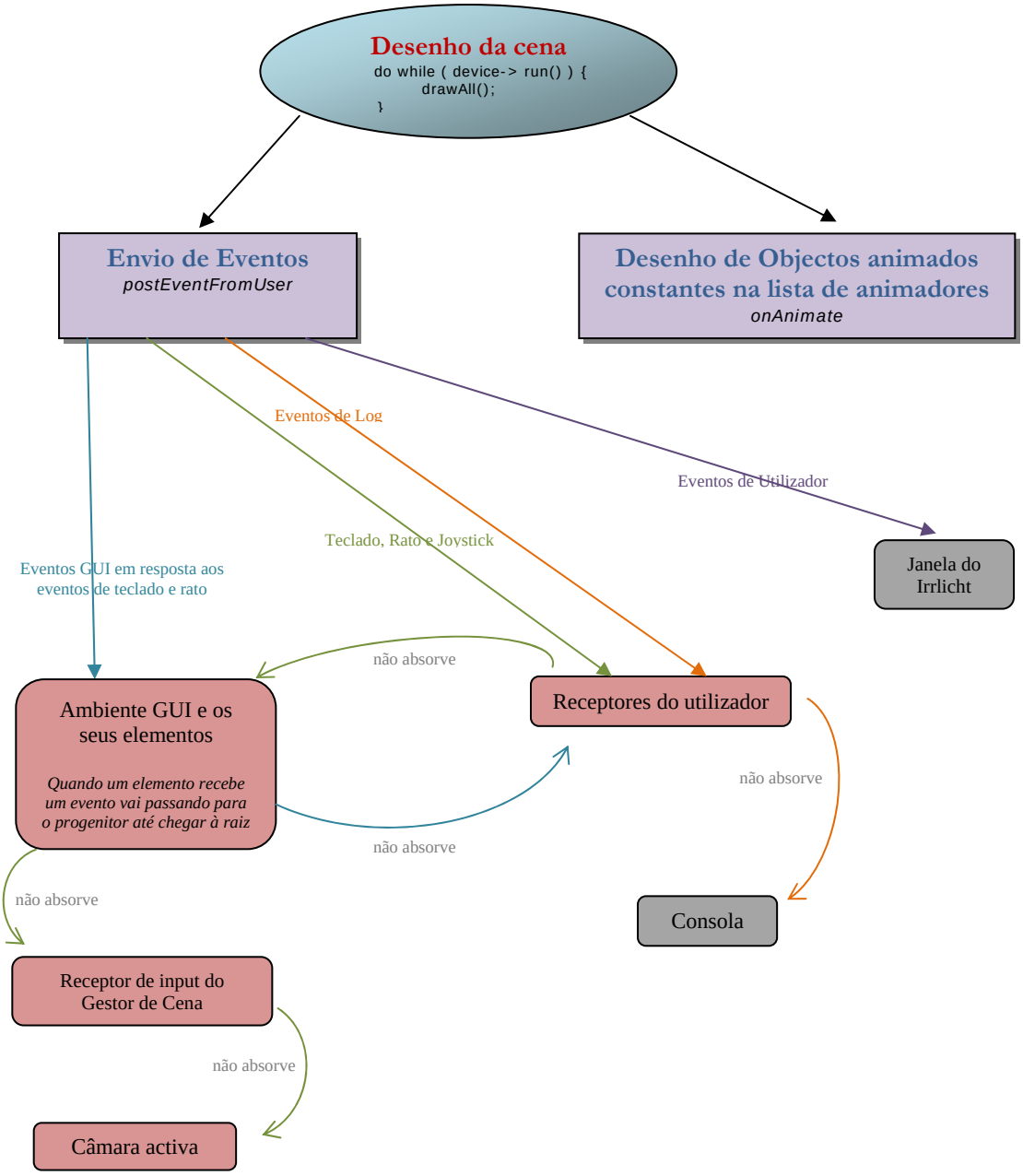

Figura 5. Envio de eventos pelo Irrlicht

O processamento de eventos no Irrlicht passa essencialmente pelo uso da classe IEventReceiver (Figura 6) que permite que um objecto receba eventos dos seguintes tipos: Teclado; Rato; Joystick; Logs; e do Utilizador.

A hierarquia descendente desta classe está definida na seguinte figura (figura 5).

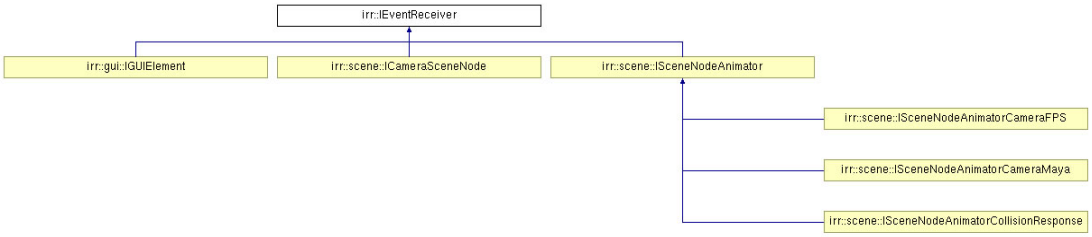

Figura 6. Hierarquia de classes no Motor de Eventos Irrlicht

Os objectos pendurados nesta hierarquia de classes, dependendo das suas características podem receber determinado tipo de eventos (por exemplo, uma câmara pode receber eventos de input). Apenas, as câmaras quando são criadas manifestam interesse em receber eventos ou não. Os receptores do tipo Utilizador, para receberem eventos têm de manifestar interesse em recebe-los através do método *setEventReceiver*. Os eventos começam usualmente na função *postEventFromUser*, que é invocada dentro do loop infinito que desenha a cena sempre que ocorre um evento, e são passados por uma cadeia de receptores de eventos, que são processados pela função *OnEvent* que devolve verdadeiro (true) ou falso (false) se é um tipo de evento do seu interesse ou não, respectivamente. A função *postEventFromUser* envia um evento. Normalmente o programador não necessita de usar este método, uma vez que é usado pelo motor internamente.

virtual bool irr::IrrlichtDevice::postEventFromUser (const SEvent & *event*)

SEvents é uma estrutura que guarda informação sobre um evento.

```
struct SEvent{
        union {
                struct SGUIEvent   GUIEvent;
                struct SJoystickEvent   JoystickEvent;
                struct SKeyInput   KeyInput;
                struct SLogEvent   LogEvent;
                struct SMouseInput   MouseInput;
                struct SUserEvent   UserEvent;
        };
        EEVENT_TYPE EventType;
};
```

A estrutura guarda o tipo de evento que foi enviado (*EEVENT_TYPE EventType*) e informação mais detalhada do evento:

Estrutura que guarda eventos da interface gráficos com o utilizador

```
struct {
        * Caller;
        gui::IGUIElement* Element;
        gui::EGUI_EVENT_TYPE EventType;
};
```

Estrutura que guarda eventos de joystick

```
struct SJoystickEvent{
        u32 ButtonStates;
        pl3D6 Axis[NUMBER_OF_AXES];
        u16 POV;
        u8 Joystick;
}
```

Estrutura que guarda eventos de log

```
struct {
        const * Text;
        ELOG_LEVEL Level;
};
```

Estrutura que guarda eventos de teclado

```
struct SKeyInput{
        wchar_t Char;
        EKEY_CODE Key;
        bool PressedDown;
        bool Shift;
        bool Control;
};
```

Estrutura que guarda eventos de rato

```
struct SMouseInput{
        s32 X;
        s32 Y;
        f32 Wheel;
        EMOUSE_INPUT_EVENT Event;
};
```

Estrutura que guarda eventos do utilizador

```
struct SUserEvent{
        s32 UserData1;
        s32 UserData2;
};
```

Onde S32 é um inteiro com sinal de 32 bits

Os tipos de eventos (*irr::EEVENT_TYPE*) podem ser os seguintes:

- *EET_GUI_EVENT* – evento do interface gráfico com o utilizador. São criados em resposta a eventos de rato ou teclado. Quando um elemento GUI recebe um evento, vai processá-lo e devolve verdadeiro, ou passa para o elemento progenitor. Se o evento não for absorvido até que atinja a raiz dos elementos GUI então é passado para os Receptores do utilizador
- *EET_MOUSE_INPUT_EVENT* – evento de rato. São criados pelo dispositivo e normalmente passados para a função *IrrlichtDevice::postEventFromUser* em resposta ao input do rato. Os eventos de rato são primeiro passados para os receptores do utilizador, depois para o

ambiente GUI e os seus elementos, depois para gestor de cena que por sua vez os passa para a câmara activa.

- *EET_KEY_INPUT_EVENT* – evento de teclado. Funcionam exactamente da mesma forma que os eventos de rato e percorrem também o mesmo caminho.
- *EET_JOYSTICK_INPUT_EVENT* – evento de joystick. Funcionam exactamente da mesma forma que os eventos de rato e de teclado. Percorrem também o mesmo caminho.
- *EET_LOG_TEXT_EVENT* – Um evento de log. Normalmente são mensagens de erros e avisos, também pode ser uma mensagem informativa. São apenas passados para os receptores de utilizador, se existir algum. Se for absorvido não irá aparecer nenhuma mensagem na consola.
- *EET_USER_EVENT* – Um evento de utilizador com dados do utilizador. Este tipo de evento não é usado pelo motor irrlicht, mas pode ser usado pelo utilizador se pretender enviar alguma informação pelo motor.

A produção de animação no irrlicht faz-se usando uma lista de objectos que são possíveis de animar. Entre eles estão as câmaras.

```
core::list<ISceneNodeAnimator*> Animators;
```

Cada objecto animado no irrlicht tem um estado que é alterado a cada frame. Cada vez que desenha a cena é enviada uma ordem de animação, com o método *OnAnimate(Time)*, a todos os objectos que estão na lista *Animators*. O método *OnAnimate(Time)* envia para cada objecto da lista a ordem *animateNode(Node, Time)* que faz com que o objecto execute a animação tendo em atenção o tempo actual.

Existem dois tipos de câmaras disponíveis para serem usadas no motor de jogos irrlicht, que se comportam de forma diferente. A câmara *CameraFPS* que funciona como sendo o utilizador o centro, olhando para os objectos à sua volta e a *CameraMaya*, ao estilo Maya, que está sempre a olhar para um objecto central.

A classe *ICameraSceneNode*, quando recebe um evento de input encaminha-o para todos os objectos constantes na lista de animadores.

```
bool CCameraSceneNode::OnEvent(const SEvent& event)
{
	if (!InputReceiverEnabled)
		return false;

	// send events to event receiving animators

	core::list<ISceneNodeAnimator*>::Iterator ait = Animators.begin();

	for (; ait != Animators.end(); ++ait)
		if ((*ait)->isEventReceiverEnabled() && (*ait)->OnEvent(event))
			return true;

	// if nobody processed the event, return false
	return false;
}
```

As classes *ISceneNodeAnimatorCameraFPS* e *ISceneNodeAnimatorCameraMaya* têm implementada a função OnEvent que processa e filtra os eventos de Input que recebem.

Relativamente a outros objectos possíveis de animar, como não pertencem à hierarquia da classe *IEventReceiver*, não reescreverem a função *onEvent*, o que significa que esses objectos por si só não absorvem os eventos. Para que estes possam receber eventos também não basta descenderem da classe *ISceneNodeAnimator*, adicionalmente terá de ser criada, pelo utilizador, uma classe receptora de eventos que subscreva o método *onEvent* para receber eventos. O exemplo seguinte mostra como podemos criar uma classe receptora de eventos de teclado e rato. Neste exemplo os eventos recebidos são também guardados.

```
class MyEventReceiver : public IEventReceiver
{
public:
        // This is the one method that we have to implement
        virtual bool OnEvent(const SEvent& event)
        {
                switch(event.EventType)
                {
                        // Remember whether each key is down or up
                        case irr::EET_KEY_INPUT_EVENT:
                                KeyIsDown[event.KeyInput.Key] = event.KeyInput.PressedDown;
                                break;
                        case irr::EET_MOUSE_INPUT_EVENT:
                                if (event.MouseInput.Event =
                                                irr::EMIE_LMOUSE_PRESSED_DOWN)
                                        MouseButton[0]=true;

                                if (event.MouseInput.Event ==
irr::EMIE_RMOUSE_PRESSED_DOWN)
                                        MouseButton[1]=true;
                                break;
                        default:
                                break;
                }
                return true;
        }

        // This is used to check whether a key is being held down
        virtual bool IsKeyDown(EKEY_CODE keyCode) const
        {
                return KeyIsDown[keyCode];
        }

        // Verifica se o botão direito ou esquerdo do rato foi pressionado
        virtual bool isMouseButtonPressed(int button)
        {
                int m=MouseButton[button];

                MouseButton[button]=false;
                return m;
        }

        MyEventReceiver()
        {
                for (u32 i=0; i<KEY_KEY_CODES_COUNT; ++i)
                        KeyIsDown[i] = false;
                MouseButton[0]=false;
                MouseButton[1]=false;

        }

private:
        // We use this array to store the current state of each key
        bool KeyIsDown[KEY_KEY_CODES_COUNT];
        // posição 0 – botão esquerdo; posição 1 – botão direito
        bool MouseButton[2];
    };
```

Suponhamos agora que se pretende desligar uma luz ao carregar numa tecla. Teríamos de, criar uma classe como a apresentada a seguir para receber eventos de teclas, e na classe referente à luz teria de ter um método para ligar e desligar a luz que seria invocado quando se carregasse na tecla correspondente. Essa verificação, da tecla quando era carregada, seria feita dentro do *loop* do programa principal através do método *IsKeyDown* da classe receptora de eventos *MyEventReceive*.

```
class MyEventReceiver : public IEventReceiver
{
public:
	// This is the one method that we have to implement
	virtual bool OnEvent(const SEvent& event)
	{
		switch(event.EventType)
		{
			// Remember whether each key is down or up
			case irr::EET_KEY_INPUT_EVENT:
				KeyIsDown[event.KeyInput.Key] = event.KeyInput.PressedDown;
				break;

			default:
				break;
		}
		return true;
	}

	// This is used to check whether a key is being held down
	virtual bool IsKeyDown(EKEY_CODE keyCode) const
	{
		return KeyIsDown[keyCode];
	}


	MyEventReceiver()
	{
		for (u32 i=0; i<KEY_KEY_CODES_COUNT; ++i)
			KeyIsDown[i] = false;

	}

private:
	// We use this array to store the current state of each key
	bool KeyIsDown[KEY_KEY_CODES_COUNT];
    };
```

Uma vez que no irrLicht não existe uma classe que permite de forma directa ligar e desligar uma luz, uma solução seria acrescentar à cena uma luz dinâmica cuja cor será preta. Desta forma, simula que a luz está desligada.

```
...
ILightSceneNode *lightnode = scenemanager->addLightSceneNode( 0, core::vector3df(0,400,-200),
video::SColorf(0.0f,0.0f,0.0f), 1.0f, 1 );

MyEventReceiver receiver;

IrrlichtDevice *device = createDevice( video::EDT_SOFTWARE, dimension2d<s32>(640, 480), 16,
                        false, false, false, &receiver);
...
while(device->run())
{
        ...
        if(receiver.IsKeyDown(irr::KEY_KEY_Q))
                        lightnode->getLightData().AmbienteColor.set(0,0,0);
        ...
}
```

2.5. Processamento de Eventos no OGRE

OGRE é motor essencialmente gráfico sendo desenvolvidas extensões que permitem que este possa comportar-se como um motor de jogo. As funcionalidades mais importantes do motor estão descritas na tabela 2.

Características gerais	O motor é completamente orientado a objectos e independente da plataforma Possui mecanismo para carregar cenas Possui mecanismos para extender o motor sem que este seja recompilado
Scripting	Carregamento da cena em múltiplos passos Carregamento de materiais
Física	Simulação básica, detecção de colisão, e animação do esqueleto
Iluminação	Por pixel, por vértice, mapeamento
Sombras	Volumes de sombras, mapeamento de sombras
Texturas	Aplicação de texturas, Multi-texturing, Bump mapping, Mipmapping
Shaders	Vertex, Pixel
Gestão da Cena	BSP, Octree Occlusion Culling, LOD
Animação	Animação de esqueleto, Inverse Kinematics, Animation Blending
malhas	Carregamento de vários tipos de malhas, Skinning, Progressive
Superficies e	Splines

curvas	
Efeitos especiais	Environment Mapping, Lens Flares, Billboarding, Particle System, Motion Blur, Sky, Water, Fog:
Rendering	Fixed-function, Render-to-Texture, Fonts, GUI

Tabela 2. Funcionalidades mais importantes do motor OGRE

Este motor possui uma classe *FrameListener* que recebe notificações de eventos de frame, isto é, início de frame, execução de frame e fim de frame. O Irrlicht não possui uma classe do género pelo que o utilizador, se precisar, é forçado a criar uma, assim como, a manter uma lista de objectos que pretendam receber eventos de frame.

Para adicionarmos um *FrameListener* a um objecto, usamos o método *addFrameListener*, que coloca o objecto numa lista para o efeito. Existe outra lista para os objectos marcados para serem removidos da lista onde constam os objectos a receber notificações de frame. No início de cada frame esses objectos são efectivamente eliminados da lista.

```
/** Set of registered frame listeners */
 std::set<FrameListener*> mFrameListeners;

 /** Set of frame listeners marked for removal*/
 std::set<FrameListener*> mRemovedFrameListeners;
```

Tudo inicia quando invocamos o método *startRendering()* que no loop infinito chama o método *renderOneFrame()* (ver Figura 7), que por sua vez envia notificações de frame para todos os objectos que estejam na lista de receptores de eventos de frame.

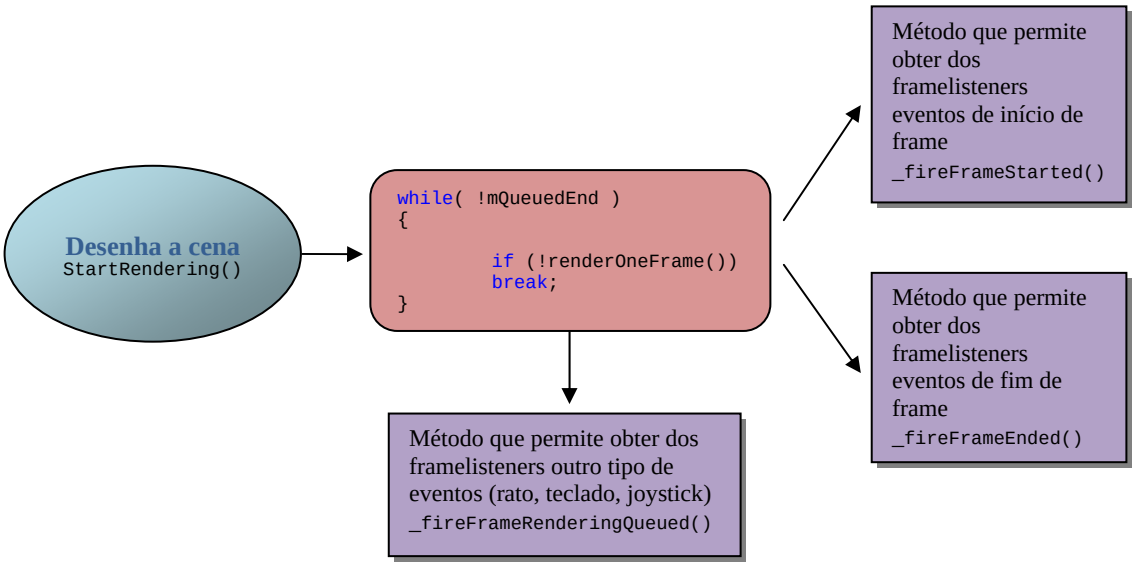

Figura 7. Envio de eventos pelo OGRE

Quando um objecto é notificado de um evento de frame é-lhe enviado como argumento, em milisegundos, o tempo que passou desde o último evento *TimeSinceLastEvent*, assim como o tempo que passou desde o último evento do mesmo tipo *TimeSinceLastFrame* (ou seja, o tempo da frame completa).

Foi desenvolvida uma biblioteca OIS especificamente para tratamento de eventos de input (teclado, rato, joystick, etc.). O método *setEventCallback(…)* adiciona um listener para o objecto que depende do tipo de eventos de input que pretende receber. Existem 3 tipos de Listeners: *KeyListener*, *MouseListener* e *JoyStickListener*. (ver Figura 8)

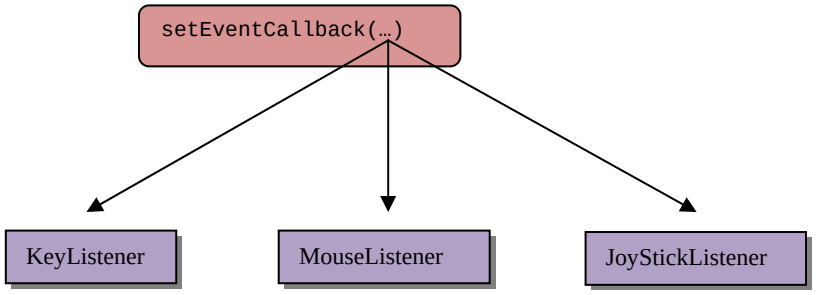

Figura 8. Listeners no OGRE

```
class _OISExport KeyListener
        {
        public:
                virtual ~KeyListener() {}
                virtual bool keyPressed( const KeyEvent &arg ) = 0;
                virtual bool keyReleased( const KeyEvent &arg ) = 0;
        };


class _OISExport MouseListener
        {
        public:
                virtual ~MouseListener() {}
                virtual bool mouseMoved( const MouseEvent &arg ) = 0;
                virtual bool mousePressed( const MouseEvent &arg, MouseButtonID id ) = 0;
                virtual bool mouseReleased( const MouseEvent &arg, MouseButtonID id ) = 0;
        };


class _OISExport JoyStickListener
        {
        public:
                virtual ~JoyStickListener() {}

                virtual bool buttonPressed( const JoyStickEvent &arg, int button ) = 0;
                virtual bool buttonReleased( const JoyStickEvent &arg, int button ) = 0;

                virtual bool axisMoved( const JoyStickEvent &arg, int axis ) = 0;

                //Joystick Event, amd sliderID
                virtual bool sliderMoved( const JoyStickEvent &, int ) {return true;}
                //Joystick Event, amd povID
                virtual bool povMoved( const JoyStickEvent &, int ) {return true;}
        };
```

Estes listeners usam como argumentos nos seus métodos, eventos da seguinte hierarquia de classes da Figura 9:

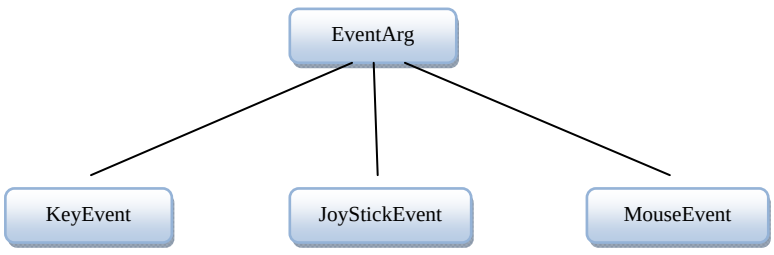

Figura 9. Arquitectura de classes para armazenar a informação dos eventos no OGRE

EventArg é a classe base para todos os eventos. *KeyEvent* é uma classe especificamente para o tratamento de eventos de teclado. *JoyStickEvent* e *MouseEvent* são classes especificamente para o tratamento de eventos de Joystick e Rato, respectivamente.

Não existe um motor específico para controlar o fluxo de eventos. Os eventos resultam de dispositivos de input, colisão de objectos e eventos de frame. No que diz respeito à colisão dos objectos esta é feita por exemplo pelo motor de física ODE, através de métodos que determinam se existe colisão entre pares, usando o método *dCollide*, por exemplo, e desta forma definir o que fazer se detectar que existe colisão. Para que um objecto possa receber eventos terá de descender dos listeners disponíveis, nomeadamente *Framelisteners*, *MouseListeners*, *KeyListeners* e *JoystickListener* e manifestar interesse em receber eventos desse tipo. Desta forma, os objectos não comunicam directamente uns com os outros e um evento não tem um destinatário específico, nem é possível definir um.

Recorrendo a um exemplo utilizado anteriormente, suponhamos que se pretende acender uma luz quando carregamos numa tecla. Teríamos de criar uma classe descendente de *KeyListener* e reescrever os métodos *KeyPressed* e/ou *KeyReleased*. Recebíamos os eventos de teclado e ordenávamos à luz que acendesse ou apagasse, usando um método da classe luz para esses efeitos, respectivamente.

Fazendo uma análise comparativa, podemos inferir que ambos os motores possuem uma filosofia no que diz respeito ao processamento de eventos muito próxima, com excepção das câmaras que tratam os eventos de forma diferente, para além de realizar de forma integrada a detecção de colisão. O irrLicht define 3 tipos de câmaras diferentes que já possuem de forma predefinida toda a interacção com a câmara. O Ogre trata a câmara como um objecto usual que para receber eventos tem de possuir uma classe receptora de eventos e a detecção de colisão não faz parte do seu núcleo (core). Ambos não possuem um mecanismo direccionado para a gestão de eventos, nem permitem que os objectos comuniquem entre si directamente. Salienta-se apenas que os eventos percorrem um caminho diferente em cada um dos motores.

Capítulo 3 – DESENVOLVIMENTO

Como já foi referido anteriormente, o trabalho realizado no âmbito desta tese é na perspectiva de acrescentar funcionalidade e interacção ao ambiente 3D do Curitiba. Foi efectuado um estudo e implementação de uma arquitectura de um motor de eventos, estruturado segundo a filosofia do Curitiba e da programação orientada a objectos. Um dos requisitos deste projecto é permitir que o fluir dos eventos possa ser especificado nos ficheiros de projecto, que inicialmente se limitavam a uma descrição de uma cena estática e não interactiva, com recurso à notação XML.

De salientar que, o objectivo deste projecto não é, no entanto, criar um sistema de animação complexo, mas sim o desencadear da sequência de eventos que despoletam e determinam o tempo de vida, se for caso disso, das animações e mudanças de estado.

Por exemplo, podemos definir que quando a câmara entra num quarto se acende uma luz, que por sua vez acorda os mosquitos que irão rodopiar em torno da mesma. Não se pretende que o motor de eventos ligue ou desligue uma luz, muito menos que faça animação de mosquitos. Pretende-se, isso sim, definir o fluxo de eventos.

O motor de eventos deve-se pautar pela simplicidade, sem no entanto sacrificar a expansibilidade. Desta forma espera-se obter uma boa curva de aprendizagem para um programador familiarizado com a API do Curitiba.

Resumidamente, procura-se definir uma arquitectura de um motor de eventos, e uma forma de especificação dos mesmos, que seja ao mesmo tempo simples e expansível.

3.1. Envio de Eventos – Tipos de dados

O motor de eventos implementado permite o envio e recepção de mensagens sob a forma de dados associados ao tipo de evento.

Esses dados são representados pela classe *IEventData*. Esta classe é conceptualmente uma interface. Desta descendem várias classes concretas já implementadas. (Figura 10)

Relativamente às já existentes:

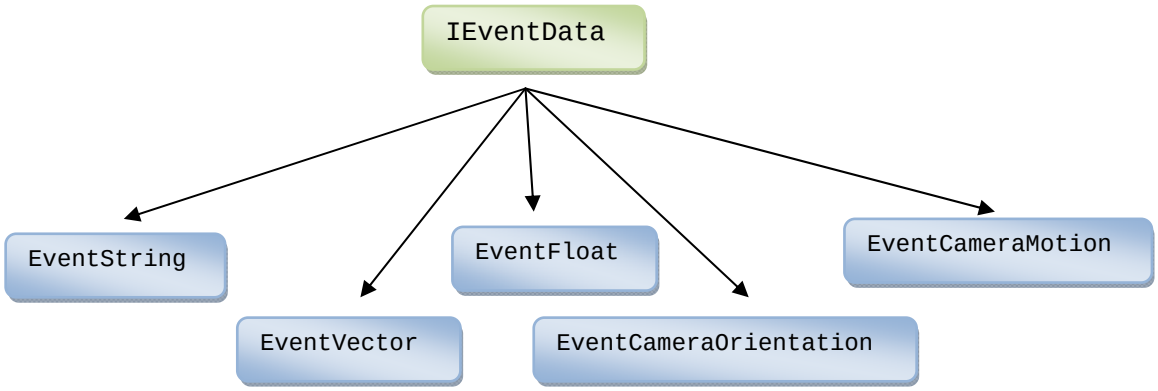

Figura 10. Arquitectura de classes para armazenar os dados do motor de eventos do Curitiba

Certas classes representam tipos básicos de dados como é o caso de *EventString*, *EventFloat*, ou *EventVector*. Outras, como por exemplo *EventCameraOrientation* e *EventCameraMotion*, já contêm uma colecção de valores.

Por exemplo, *EventCameraOrientation* é uma classe composta por um conjunto de atributos necessários para permitir calcular a orientação da câmara.

```
float alpha;
float beta;
float newX;
float newY;
float oldX;
float oldY;
float scaleFactor;
```

Embora o recurso a *strings* permita praticamente passar todo o tipo de informação desejado, a sua utilização obriga a um processamento que pode ser ineficiente de um ponto de vista computacional.

Caso, no contexto de uma nova aplicação, se torne necessário outro tipo de dados pode-se definir uma nova classe que respeite a interface exposta em *IEventData*, evitando assim o custo associado à interpretação de *strings*.

Respeitando a filosofia do Curitiba, a criação de novos eventos é realizada recorrendo a uma *factory*. Para tal foi criada também uma classe *EventFactory*, que

permite criar (fabricar) qualquer uma das classes descendentes de *IEventData* já definidas, usando sempre o mesmo método, neste caso o método *create*, onde se passa como argumento o tipo de objecto que se pretende criar.

```
static curitiba::event_::IEventData* create (std::string type);
```

3.2. Arquitectura do Motor de Eventos

Após a pesquisa e análise efectuada, depreende-se que a utilização de *Listeners* para a gestão de eventos obedece aos requisitos deste projecto.

Desta forma, começou-se pela implementação de um conjunto de classes que permitem a criação de *Listeners* genéricos, i.e. sem semântica associada. Está abordagem contrasta com a usualmente seguida nos motores gráficos/jogos, em que pelo menos alguns eventos têm uma semântica clara associada. No entanto ao optar pela abordagem despida de semântica não existe qualquer perda de funcionalidade, e ganha-se em simplicidade do motor de eventos.

O conceito é simples, basicamente funciona com o envio da mensagem, com os dados referentes ao evento ocorrido, por parte dos objectos, sendo a classe *EventManager* responsável pela gestão dessas mensagens. Os objectos que pretendam receber mensagens de um determinado tipo (*Listeners*) registam o seu interesse na classe *EventManager*.

A classe *EventManager* é portanto responsável por registar (adicionar) e remover os objectos como *listeners* de determinados eventos, assim como, sempre que recebe uma notificação de um evento, filtrar e reencaminhar para os *Listeners* correspondentes.

Para isso, usa um vector do tipo:

```
map<std::string eventType, vector<Listener *>> listeners;
```

O seguinte exemplo, em forma de esquema, mostra como está organizada a informação da classe EventManager:

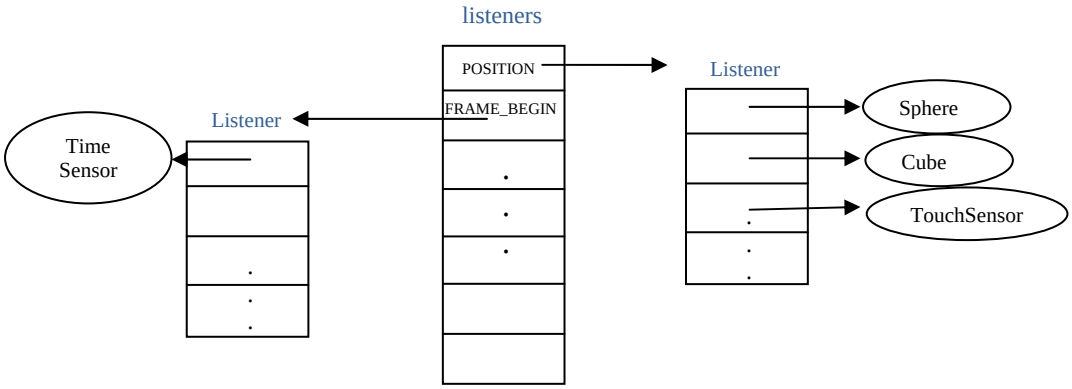

Figura 11. Estrutura de classes para a gestão de eventos do Curitiba

Neste caso, temos três objectos registados como querendo receber eventos de *POSITION*, que são *Sphere*, Cube e *TouchSensor*. Existe um objecto registado como querendo receber eventos de *FRAME_BEGIN*, que é o *TimeSensor*.

Todos os objectos podem enviar mensagens e numa fase inicial do projecto estas indicavam apenas qual o evento que ocorreu e um conjunto de dados adicionais referentes a esse mesmo evento.

```
notifyEvent(char *eventType, IEventData *evt)
```

O evento pode ser um já existente como *POSITION* ou *FRAME_BEGIN*, assim como um qualquer evento criado pelo utilizador.

O esquema a seguir apresentado mostra como esse envio e reencaminhamento de mensagens é processado.

Exemplo: Suponhamos que o Objecto1 envia o evento X. Podemos verificar que os objectos 2, 3 e 4 estão registados para receber eventos X e portanto sempre que este ocorre os objectos 2, 3 e 4 recebem-no. Existe também um envio de evento Y que funciona exactamente seguindo o mesmo critério. Os objectos 2 e 3 registaram-se para recebe-los. (Figura 12)

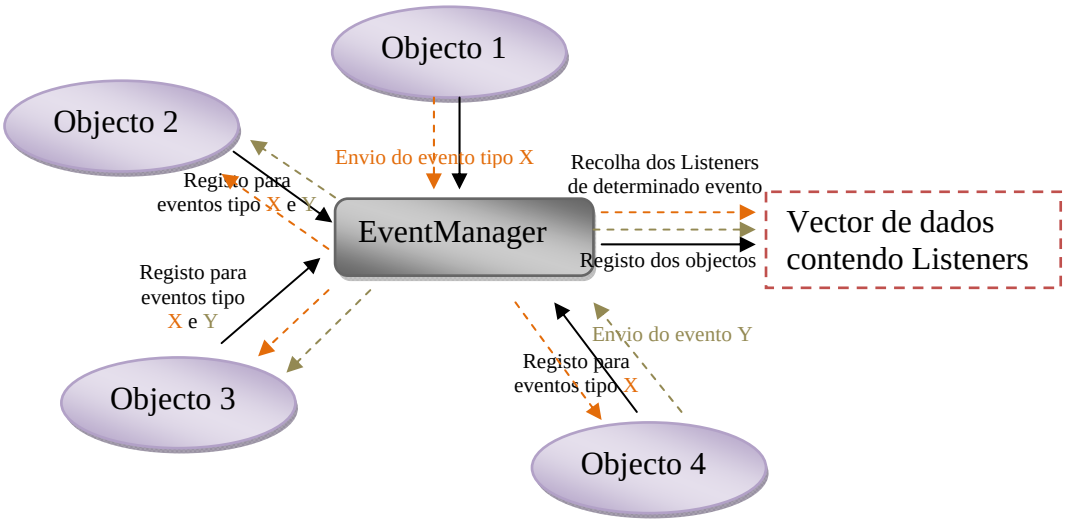

Figura 12. Esquematização da 1ª abordagem da circulação de eventos no Motor de Eventos do Curitiba

Instanciando um problema concreto, podemos supor que se carregarmos numa tecla, é enviado um evento para ligar as luzes (*LIGHT_ON*) e se carregar noutra tecla envia um evento para desligar as luzes (*LIGHT_OFF*). Logo todas as luzes que se registam para receber estes eventos serão apagadas e ligadas com este sistema.

Sempre que se pressionar a tecla *i* ou *u* será enviada uma notificação de evento com o seguinte excerto de código:

```
curitiba::event_::IEventData *e=curitiba::event_::EventFactory::create("String");

if(event.GetKeyCode()=='U' || event.GetKeyCode()=='u'){
        EVENTMANAGER->notifyEvent("LIGHT_OFF", e);
}

if(event.GetKeyCode()=='I' || event.GetKeyCode()=='i'){
        EVENTMANAGER->notifyEvent("LIGHT_ON", e);
}
```

Teríamos de criar uma classe *LightWithSwitch*, subclasse de *Light*, que teria de possuir um método para registo como *listener*, invocado na criação desta instância.

```
void LightWithSwitch::addLightListener(void)
{
        curitiba::event_::Listener *lst;
        lst=this;
        EVENTMANAGER->addListener("LIGHT_ON",lst);
        EVENTMANAGER->addListener("LIGHT_OFF",lst);
}
```

Por outro lado, implementa o método que permite processar esses eventos.

```
void LightWithSwitch::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
        if (strcmp(eventType, "LIGHT_ON")==0)
                lightOn();
        if (strcmp(eventType, "LIGHT_OFF")==0)
                lightOff();
        }
}
```

Por sua vez, os métodos *lightOn* e *lightOff* ligam e desligam a luz respectivamente.

Suponhamos que pretendemos agora ligar e desligar duas luzes específicas na sala usando duas teclas.

Exemplo: Sempre que se pressionar a tecla *i* ou *u* serão enviadas duas notificação de evento da seguinte forma:

```
curitiba::event_::IEventData *e=curitiba::event_::EventFactory::create("String");

if(event.GetKeyCode()=='U' || event.GetKeyCode()=='u'){
        EVENTMANAGER->notifyEvent("LIGHT_OFF_LIVINGROOM1", e);
        EVENTMANAGER->notifyEvent("LIGHT_OFF_ LIVINGROOM2", e);
}

if(event.GetKeyCode()=='I' || event.GetKeyCode()=='i'){
        EVENTMANAGER->notifyEvent("LIGHT_ON_LIVINGROOM1", e);
        EVENTMANAGER->notifyEvent("LIGHT_ON_LIVINGROOM1", e);
}
```

Teríamos de criar uma classe *LightWithSwitchLivingRoom1 e outra LightWithSwitchLivingRoom2* que teriam de possuir um método para registo como *listener*.

```
void LightWithSwitchLivingRoom1::addLightListener(void)
{
        curitiba::event_::Listener *lst;
        lst=this;
        EVENTMANAGER->addListener("LIGHT_ON_LIVINGROOM1",lst);
        EVENTMANAGER->addListener("LIGHT_OFF_LIVINGROOM1",lst);
}
```

```
void LightWithSwitchLivingRoom2::addLightListener(void)
{
        curitiba::event_::Listener *lst;
        lst=this;
        EVENTMANAGER->addListener("LIGHT_ON_LIVINGROOM2",lst);
        EVENTMANAGER->addListener("LIGHT_OFF_LIVINGROOM2",lst);
}
```

Por outro lado, é necessário implementar os métodos que permitem processar esses eventos em cada uma das classes.

```
void LightWithSwitchLivingRoom1::eventReceived(char *eventType, curitiba::event_::IEventData
*evt){
        if (strcmp(eventType, "LIGHT_ON_LIVINGROOM1")==0)
                lightOn();
        if (strcmp(eventType, "LIGHT_OFF_LIVINGROOM1")==0)
                lightOff();
        }
}
```

```
void LightWithSwitchLivingRoom2::eventReceived(char *eventType, curitiba::event_::IEventData
*evt){
        if (strcmp(eventType, "LIGHT_ON_LIVINGROOM2")==0)
                lightOn();
        if (strcmp(eventType, "LIGHT_OFF_LIVINGROOM2")==0)
                lightOff();
        }
}
```

Este processo implica a criação de uma classe para cada luz específica. Assim pode-se interagir individualmente com uma luz.

Vamos tentar agora construir um cenário um pouco mais complexo. Agora pretendemos que a luz da sala seja automática e ligue assim que a câmara entre na sala, desligando-se quando a câmara sair.

Teremos de criar uma nova subclasse de *Light* com os atributos correspondentes à dimensão da caixa e com um ponto no centro da caixa. Por outro lado, essa classe teria de se registar como *listener* de eventos do tipo "*POSITION*", o evento que a câmara envia quando muda de posição. Vejamos como poderíamos implementar este exemplo.

Construímos uma classe *LightWithSensor*, que descende de *Light* e teria de possuir mais dois atributos representando a caixa envolvente e o centro.

```
curitiba::math::vec3 box
curitiba::math::vec3 center
```

Precisaria de um método para registo como *listener*.

```
void LightWithSensor::addLightListener(void)
{
        curitiba::event_::Listener *lst;
        lst=this;
        EVENTMANAGER->addListener("POSITION",lst);
}
```

Por outro lado, implementa o método que permite processar esses eventos.

```
void LightWithSensor::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
    if(isNear(evt->getData()))
        if (enabled==false) lightOn();
    else
        if (enabled==true) lightOff();
}
```

Cujo método *isNear* verifica que entrou na caixa.

O conceito implementado consiste numa luz activada por um sensor de proximidade. Um sensor deste tipo poderia ser utilizado em muitas outras circunstâncias, como por exemplo detecção de colisões entre objectos, logo seria natural que existisse uma classe própria para o implementar.

Podemos fazer a separação desta classe em duas e obter os mesmos resultados. Utilizar-se-ia a classe *LightWithSwitch* já implementada para a luz, e um sensor de proximidade, que recebesse eventos da câmara e enviasse eventos para a luz, assumindo o sensor a tarefa de reencaminhar o evento. Neste contexto, a câmara envia um evento de posição, o sensor recebe-a e verifica se está dentro da sua área, se estiver envia um evento a notificar a luz que consequentemente se liga. O mesmo processo é aplicado para a luz se desligar quando se afasta.

Criamos um sensor de proximidade *ProximitySensor* que se regista como receptor de eventos da câmara.

```
void ProximitySensor::addProximityListener(void){
    curitiba::event_::Listener *lst;
    lst = this;
    EVENTMANAGER->addListener("POSITION",lst);
 }
```

Por outro lado, implementa o método que permite processar esses eventos.

```
void ProximitySensor::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
    curitiba::event_::IEventData *e=curitiba::event_::EventFactory::create("String");

    if(isNear(evt->getData())){
        if (enabled==false){
            EVENTMANAGER->notifyEvent("LIGHT_ON", e);
            enabled=true;
        }
    }else{
        if (enabled==true){
            EVENTMANAGER->notifyEvent("LIGHT_OFF", e);
            enabled=false;
        }
    }
}
```

Agora suponhamos que queremos aplicar a mesma interacção às várias divisões de uma casa. Neste caso teríamos de possuir *n* classes de sensores e o mesmo número de classes de luzes. Isto porque o conjunto das luzes de cada divisão teria de receber um evento específico.

```
void LightWithSwitch<i>::addLightListener(void)
{
    curitiba::event_::Listener *lst;
    lst=this;
    EVENTMANAGER->addListener("LIGHT_ON_ROOM<i>",lst);
    EVENTMANAGER->addListener("LIGHT_OFF_ROOM<i>",lst);
}
```

Por outro lado, implementa o método que permite processar esses eventos.

```
void LightWithSwitch<i>::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
    if (strcmp(eventType, "LIGHT_ON_ROOM<i>")==0)
        lightOn();
    if (strcmp(eventType, "LIGHT_OFF_ROOM<i>")==0)
        lightOff();
    }
}
```

Criamos *n* classes para os sensores de proximidade, *ProximitySensori*, $0 >= i > n$, que se registam como receptores de eventos da câmara.

```
void ProximitySensor::addProximityListener(void){
        curitiba::event_::Listener *lst;
        lst = this;
        EVENTMANAGER->addListener("POSITION",lst);
 }
```

Por outro lado, implementam o método que permite processar este evento.

```
void ProximitySensori::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
     curitiba::event_::IEventData *e=curitiba::event_::EventFactory::create("String");

        if(isNear(evt->getData())){
                if (enabled==false){
                        EVENTMANAGER->notifyEvent("LIGHT_ON_ROOMi", e);
                        enabled=true;
                }
        }else{
                if (enabled==true){
                        EVENTMANAGER->notifyEvent("LIGHT_OFF_ROOMi", e);
                        enabled=false;
                }
        }
}
```

Todas as classes para os sensores são equivalentes, à excepção dos evento que geram, logo estes poderiam ser um parâmetro da instância, situação que obriga a implementar uma única classe para sensores de proximidade. O mesmo acontece com as classes *LightWithSwitchi*, mas neste caso para o evento recebido.

Por outro lado o próprio receptor do evento poderia ser parametrizável, evitando assim a criação múltiplos eventos cuja funcionalidade é exactamente a mesma, ligar e desligar luzes.

De forma similar, deveria ser possível que um objecto, ao registar-se como *listener*, indicasse quais os objectos dos quais pretendia receber eventos, isto para além dos eventos que pretende receber.

Ao receber um determinado evento também poderá ser útil saber quem enviou esse mesmo evento, permitindo assim um processamento do evento diferente consoante o objecto que gera o evento.

Nesta fase passa então a ser permitido especificar quem envia a mensagem e quem vai recebe-la. Quando um evento é enviado por um objecto esse evento é composto por quatro argumentos que são: o tipo de evento; quem envia o evento; quem vai receber o evento; e os dados enviados para esse evento:

```
notifyEvent(char *eventType, char *sender, char *receiver, IEventData *evt)
```

Os eventos são reencaminhados, pela classe responsável por gerir os eventos, que irá reencaminhá-los para os objectos que estão registados como interessados em receber esses eventos e que são destinatários dessas mensagens. Se a mensagem possui um destinatário concreto, envia apenas para o destinatário, caso contrário, envia para todos. Uma vez recebida a mensagem o objecto pode usar os dados nela contidos, e a informação do remetente, de forma apropriada.

Para um melhor esclarecimento, vamos exemplificar através do esquema que se segue os caminhos a percorrer no envio de mensagens de eventos do motor. Suponhamos que o objecto 1 gera o evento X destinado ao objecto 3. Podemos verificar que apesar de estarem registados os objectos 2, 3 e 4, apenas o objecto 3 recebeu o evento. No entanto existe também um envio de evento Y, que como não tem destinatário específico é enviado para todos os objectos registados. (Figura 13)

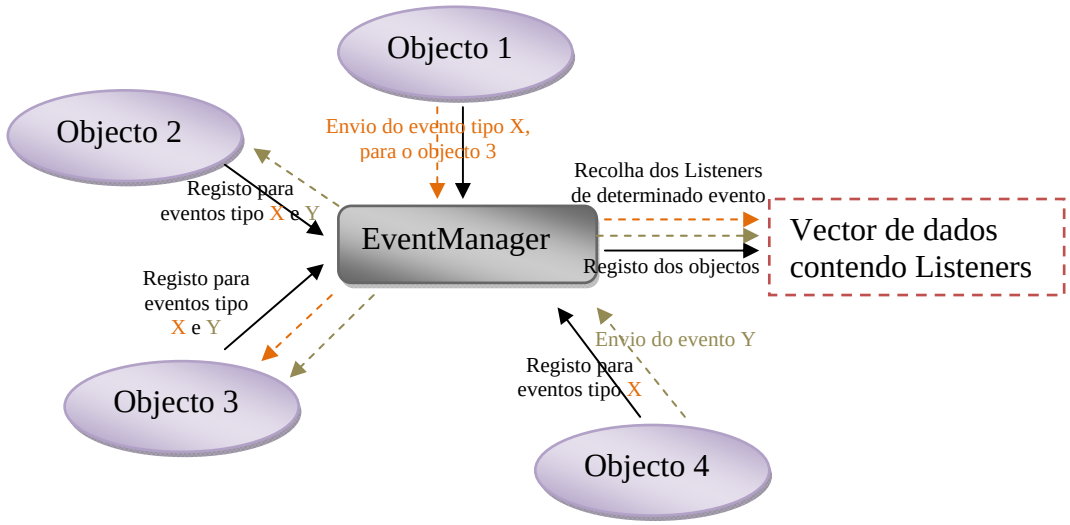

Figura 13. Esquematização da 2ª abordagem da circulação de eventos no Motor de Eventos do Curitiba

Instanciando a um problema concreto, e seguindo um dos exemplos dado anteriormente pretendemos então que a luz da sala seja ligada e desligada, assim como, a do quarto mas de forma independente. Desta forma, sempre que enviarmos uma notificação de evento vamos especificar o destinatário.

Vejamos como poderíamos expandir o motor de eventos para permitir desta forma ligar e desligar a luz do quarto usando as teclas *i* e *u*, respectivamente. Sempre que se pressionasse a tecla *i* ou *u* seria enviada uma notificação de evento para o objecto *roomLight* previamente criado.

```
curitiba::event_::IEventData *e=curitiba::event_::EventFactory::create("String");

if(event.GetKeyCode()=='U' || event.GetKeyCode()=='u'){
	EVENTMANAGER->notifyEvent("LIGHT_OFF", "keyboard", "roomLight", e);
}

if(event.GetKeyCode()=='I' || event.GetKeyCode()=='i'){
	EVENTMANAGER->notifyEvent("LIGHT_ON", "keyboard", " roomLight", e);
}
```

De salientar que esta classe *LightWithSwitch* não funciona especificamente para a luz da sala mas pode ser usada para qualquer luz de qualquer compartimento da casa uma vez que o *EventManager*, sabendo quem é o destinatário da notificação de evento, apenas encaminha o evento para o seu destino específico.

Pegando num dos exemplos anteriores, suponhamos que pretendemos ligar uma determinada luz quando a câmara se aproxima dessa luz.
Teríamos de utilizar a classe *LightWithSwitch*, que já possui um método para registo como *listener*. Usando este classe criávamos uma instância denominada por exemplo *roomLight.*

Criávamos, também, um sensor de proximidade *ProximitySensor* que se regista como receptor de eventos da câmara. Este registo passa a ser realizado para um evento que passa a ser um parâmetro da classe. Neste caso, ao instanciar a classe *ProximitySensor* é necessário especificar que o evento recebido será POSITION. Será também necessário especificar quais os eventos a enviar que são LIGHT_ON e LIGHT_OFF, e ainda, que o receptor dos eventos é, neste caso concreto, a instância da luz *roomLight.*

Seguindo este raciocínio só é necessária uma classe *ProximitySensor* e uma classe *LightWithSwitch*. Com as inicializações devidas, conseguimos realizar o fluxo de eventos desejado. (Figura 14)

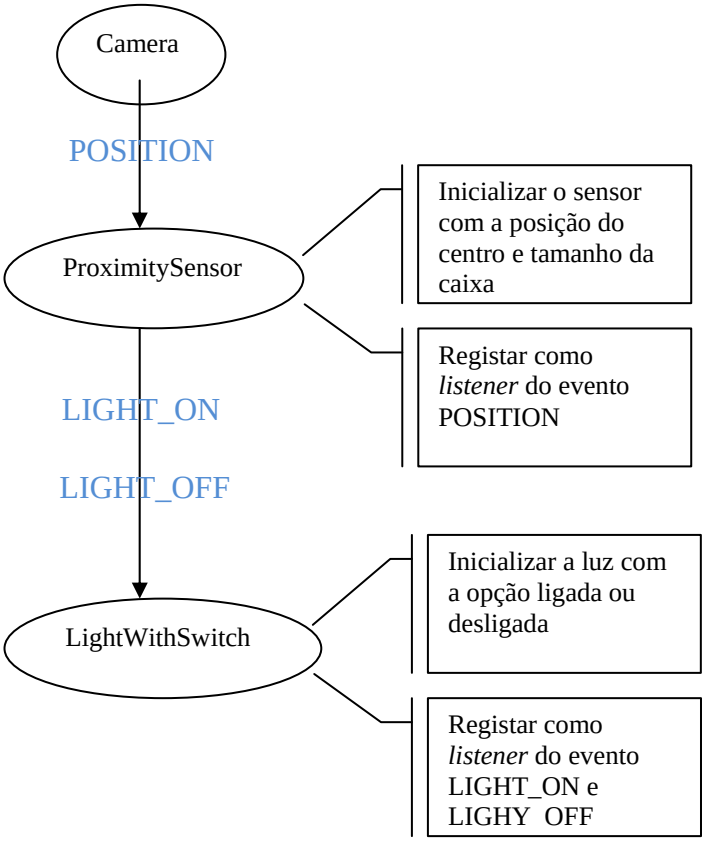

Figura 14. Encadeamento de Eventos

A definição do fluxo de eventos pode ainda ser mais simplificada se criarmos no motor de eventos a possibilidade de enviar eventos só a objectos que especifiquem querer receber o evento de um determinado remetente. Na estratégia anterior, um evento sem destinatários específicos seria enviado a todos os objectos que se registassem para receber eventos de um determinado tipo. Adiciona-se agora um campo, *restricted*, que permite, ao enviar um evento, especificar se este deve ser difundido para todos os que se registem para receber esse evento, sem especificar o remetente, ou somente aos objectos que manifestem o interesse de receber aquele evento específico daquele objecto particular.

Neste contexto, o objecto *Camera* envia eventos *POSITION* sempre que a sua posição se altera especificando que esse evento só deverá ser enviado a quem manifestar interesse em receber este evento. O *ProximitySensor* regista-se como interessado em receber eventos *POSITION*, especificamente do objecto *Camera*. Por sua vez o *ProximitySensor* gerará eventos do tipo *LIGHT_ON* e *LIGHT_OFF*, sem ter

necessidade de indicar a quem se destinam, mas especificando que são eventos restritos, ou seja, só devem ser reencaminhados a quem manifeste o interesse de receber esses mesmos eventos daquele sensor específico. Por sua vez cada luz regista-se como querendo receber eventos do tipo *LIGHT_ON* e *LIGHT_OFF* do seu respectivo sensor, ou seja do sensor cuja área inclui as luzes.

3.3. Routes

A estratégia apresentada permite criar cenários complexos de fluxo de eventos, com a criação de classes genéricas e parametrizáveis.

No entanto talvez seja mais natural que um sensor de proximidade gere sempre o mesmo evento, por exemplo "*ENTER*" e "*EXIT*", ou "*IN*" e "*OUT*". Na primeira opção só são gerados eventos quando a posição recebida implica uma saída ou entrada na zona delimitada pelos dados do sensor, enquanto que na segunda o sensor geraria eventos sempre que recebe uma posição. A primeira opção é portanto mais económica de um ponto de vista computacional, embora implique que o sensor tenha um estado, e assuma uma inicialização correcta desse mesmo estado.

Desta forma elimina-se a necessidade de especificar os eventos que serão gerados por um determinado objecto. O objecto envia os eventos que são naturais no seu contexto.

Levanta-se aqui um problema: estabelecer a ligação entre o evento *ENTER* gerado pelo sensor de proximidade e o evento *LIGHT_ON* necessário para activar a luz.

Torna-se necessário criar um mecanismo que crie uma interligação entre os objectos de forma independente dos eventos que estes enviam ou recebem, assim como a possibilidade de transformar eventos e que ao mesmo tempo permita às classes enviar e receber eventos que façam parte da sua essência. Logo, com inspiração no modelo de gestão de eventos do VRML, foi criada uma classe *Route* que vai de encontro ao que se pretende, trazendo ainda mais generalidade e abstracção entre os objectos, podendo estes funcionar como ilhas isoladas. Os objectos enviam e recebem eventos com independência, sem restrições impostas pelos eventos que os outros objectos definem, e que depois vão ser interligados através deste mecanismo, os *Routes*.

Um *Route* é um objecto genérico, não gráfico, que realiza o reencaminhamento de mensagens, ou seja permite definir uma ponte entre o emissor e receptor mesmo que os eventos sejam diferentes desde que os tipos de dados associados aos eventos coincidam. Os *routes* podem ser vistos com um transformador de eventos, ou seja, ao receber um

evento transforma-o num outro evento. Esta abordagem segue o estilo do VRML que assenta nesta filosofia. Desta forma, podemos ter a câmara que envia eventos *POSITION* e um sensor de proximidade que recebe eventos *PROXIMITY*. Fazemos a ponte entre os dois usando um *Route* que sempre que recebe um evento de *POSITION* do objecto *Camera* envia um evento *PROXIMITY* para um sensor específico. Por seu lado um *ProximitySensor* envia mensagens "*ENTER*" e "*EXIT*", respectivamente quando a posição recebida no evento "*PROXIMITY*" revele uma entrada ou saída, respectivamente, da área definida no sensor. Dois novos *Routes* estabelecem a ponte entre os eventos "*ENTER*" e "*EXIT*" do sensor e os eventos "*LIGHT_ON*" e "*LIGHT_OFF*" de luzes específicas.

O conceito de *Route* vem solucionar os problemas apresentados anteriormente, pois simplifica o fluxo dos eventos. O fluxo passa a ser independente de quem gera ou recebe eventos, sendo o fluxo especificado nos *Routes*. Ou seja, as classes *ProximitySensor* e *LightWithSwitch* não têm de ser instanciadas de forma particular pois limitam-se a dizer que pretendem receber enviar e receber determinados eventos. Cabe aos *routes* estabelecer a ponte entre estes objectos. Por exemplo, considerando o cenário da luz, *roomLight*, que é accionada por um sensor de proximidade, *sensorRoom1*, que por sua vez é accionado pela posição da câmara, *camera*, poderíamos especificar o seguinte fluxo de eventos:

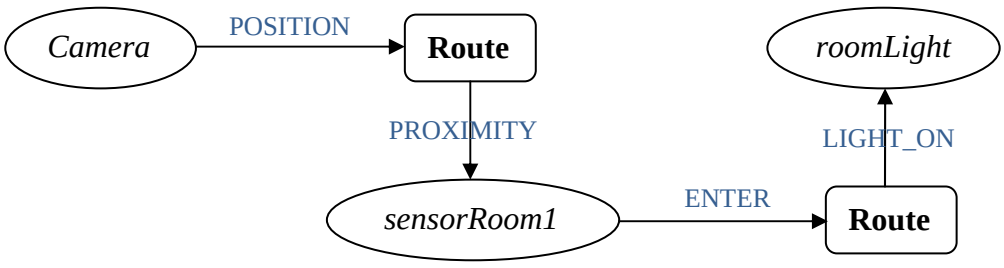

Figura 15. Acção dos *Routes* no fluxo de Eventos

```
ROUTE camera.POSITION TO sensorRoom1.PROXIMITY
ROUTE sensorRoom1.ENTER TO roomLight.LIGHT_ON
ROUTE sensorRoom1.EXIT TO roomLight.LIGHT_OFF
```

Neste contexto, os eventos gerados pela câmara e pelos sensores passam a ser restritos, ou seja, só serão reencaminhados pelo motor de eventos aos objectos que, ao registarem o seu interesse no motor de eventos especifiquem, para além do tipo de evento, o remetente. Sendo assim a criação de um novo sensor só implica a definição da sua área de actuação, e a criação de uma câmara não necessita de nenhum cuidado relativamente aos eventos por si gerados. Temos assim classes mais genéricas e simples de implementar.

Cabe aos *routes* a tarefa de definir o fluxo de eventos.

A classe *Route* é composta pelos seguintes atributos:

```
std::string sender;
std::string receiver;
std::string eventIn;
std::string eventOut;
```

A implementação dos *Routes* não implica nenhuma alteração ao motor de eventos, o que demonstra a sua flexibilidade. Os *Routes* são objectos que recebem e enviam mensagens como qualquer outro objecto. A única diferença é os parâmetros da sua instanciação que necessita de atribuir os valores apropriados aos campos enunciados acima.

Par criar uma instância de *Route* definiu-se o método:

```
Route::Route(
        std::string sender,
        std::string receiver,
        std::string eventIn,
        std::string eventOut);
```

A classe irá definir como vão ser reencaminhados os eventos. Os *routes* registam-se para receber eventos do tipo *eventIn* de um remetente particular, e enviam eventos do tipo *eventOut* para o destinatário especificado. Os dados do evento serão passados, sem sofrer alterações, do remetente para o destinatário através do motor de eventos. O código do método de criação da instância é o seguinte:

```
curitiba::event_::Listener *lst;
lst = this;
EVENTMANAGER->addListener(&this->eventInlst, sender);
```

Sendo o código para processar o evento o seguinte:

```
void Route::eventReceived(char *eventType, curitiba::event_::IEventData *evt){
  if(eventType==eventIn)
        EVENTMANAGER->notifyEvent(&eventOut,&sender,&receiver,evt);
  }
```

Vejamos como podemos agora reescrever o código do último exemplo, usando os *routes*.

Teremos então de ligar várias luzes ao entrar num quarto. Utilizamos a classe *LightWithSwitch*, exactamente como já a conhecemos de exemplos anteriores. Usando a classe criamos duas luzes denominadas, por exemplo, *roomLight1* e *roomLight2*. De seguida teríamos de criar apenas uma classe sensor de proximidade *pSensor*.

Por fim é necessário de definir que as duas luzes se ligam quando nos aproximamos do sensor. O código para criar o fluxo de eventos pretendido é o seguinte:

```
Route(”camera”, “pSensor”, “POSITION”,”PROXIMITY”);
Route(”pSensor”,”roomLight1”, “ENTER”, “LIGHT_ON”);
Route(,”pSensor”,”roomLight1”, “EXIT”, “LIGHT_OFF”);
Route(”pSensor”,”roomLight2”, “ENTER”, “LIGHT_ON”);
Route(”pSensor”,”roomLight2”, “EXIT”, “LIGHT_OFF”);
```

3.4. Intervenção no código do Curitiba

Para testar objectivamente os efeitos desta estrutura de classes, tive de estudar, analisar, experimentar e alterar o Curitiba o que devido à sua capacidade de expansibilidade, permitiu que a implementação desta funcionalidade exigisse um mínimo de reescrita do próprio Curitiba. O motor de eventos pôde assim ser construído com um nível mínimo de intrusão no código original do Curitiba.

Algumas das alterações introduzidas estão directamente relacionadas com o envio de alguns eventos essenciais ao funcionamento do Motor de Eventos. Nomeadamente o envio dos eventos *FRAME_BEGIN* e *FRAME_END* sempre que começa e acaba uma frame, respectivamente.

Também o envio da posição da câmara é essencial, apesar de esta classe não fazer parte do núcleo (*core*) do Curitiba, por este motivo foi alterada a classe *Camera* para que envie o evento *POSITION* com a sua posição sempre que há alteração da mesma.

Também, para que qualquer objecto da cena possa receber eventos, foi necessário colocar a classe *ISceneObject* a descender da classe *Listener*.

Para o carregamento dos sensores, interpoladores e *routes* a partir de um ficheiro de configuração XML, foi necessária intervenção no sentido de guardá-los em vectores para fazer a sua gestão.

3.5. Testes ao funcionamento do Motor de Eventos

Para além das alterações introduzidas na sua estrutura, de forma a integra-la com o motor gráfico, foi necessário criar outras classes que permitissem interagir com o motor de eventos e com o Curitiba de forma a experimentar interacção.

Desta forma, foi criada uma classe *LightWithSwitch* descendente de *Light* que permite ligar e desligar uma luz, uma vez que a classe *Light* não o permitia, por inicialmente o Curitiba não ter sido desenhado com o objectivo de existir interacção.

Foi criada uma classe abstracta para Sensores (*Sensor*) e outra para Interpoladores (*Interpolator*). Foram criados dois sensores descendentes da classe *Sensor* e um interpolador de posição descendente da classe *Interpolator*. (Figura 16 e 17). Quanto aos sensores um é de proximidade (*ProximitySensor*), e outro de tempo (*TimeSensor*).

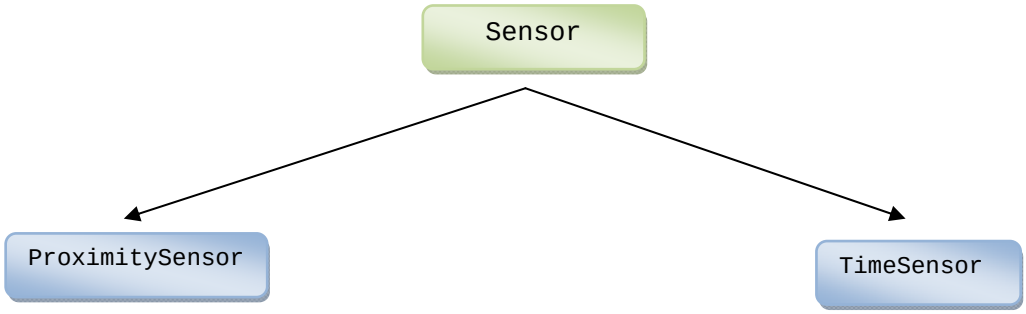

Figura 16. Hierarquia de classes que definem a organização dos sensores

ProximitySensor é um sensor que funciona como uma caixa em que se um objecto se aproxima este activa e ao afastar este desactiva. Contém como atributos o nome, se está activo ou não, o centro e o tamanho das arestas. Recebe como eventos *PROXIMITY*

como dados uma posição e envia eventos *ENTER* e *EXIT* com o dado relativo ao tempo que entrou na caixa e saiu, respectivamente.

```
std::string name;
bool enabled;
curitiba::math::vec3 center;
curitiba::math::vec3 size;
```

TimeSensor é um sensor que simula um cronómetro que dado um intervalo de tempo começa a contar e a contabilizar a fracção de tempo que passou desde o início do intervalo até ao final. Contém como atributos o nome, se está activo ou não quando começa a contar, quanto tempo conta, e se ao acabar recomeça do início ou não.

```
std::string name;
bool enabled;
int secondsToStart;
int cycleInterval;
bool loop;
```

Recebe eventos *ACTIVATE*, *DEACTIVATE* e. Este sensor está sempre a receber eventos *FRAME_BEGIN* para actualizar o seu estado. Quando recebe o evento *ACTIVATE* activa e começa a fazer a contagem depois de terem passado *secondsToStart*. O evento *DEACTIVATE* desactiva o *TimeSensor*. Envia eventos de *TIMESENSOR_IS_ACTIVE* com o tempo em que ficou activo, *TIMESENSOR_IS_FINISHED* com o tempo em que terminou e *TIMESENSOR_FRACTION_CHANGED*, com a fracção percorrida no intervalo *cycleInterval*.

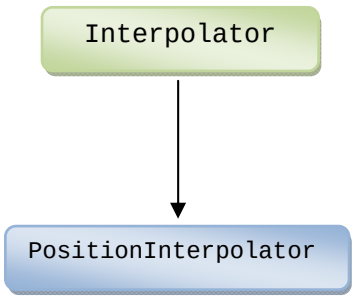

Figura 17. Hierarquia de classes que definem a organização dos interpoladores

PositionInterpolator é um interpolador que permite criar animação de posição. Tem como atributos o nome, o objecto que se pretende animar, um *array* com valores que vão de 0 a 1, e um *array* de vectores tridimensionais (*vec3*) com as posições por onde passa o objecto, o número de elementos deste vector deverá ser igual ao vector anterior pois existe correspondência directa entre as posições e as fracções percorridas que indica o *array* anterior.

```
std::string name;
ISceneObject *object;
vector<float> key;
vector<curitiba::math::vec3> keyValue;
```

Recebe eventos de *INTERPOLATOR_FRACTION_CHANGED* com a fracção correspondente à animação da posição. Envia eventos de *INTERPOLATOR_POSITION* com a correspondente posição no momento.

Exemplo 1: Suponhamos que pretendemos ligar uma luz quando nos aproximamos de um objecto localizado na origem e ao afastar desliga essa luz. No exemplo que se segue criamos um sensor de proximidade na origem com tamanho 60 nos três eixos. Inicialmente o sensor está inactivo e só irá activar quando a *MainCamera* se aproxima do sensor, como define o *Route RAproxima*. Quando o sensor activa liga a luz *Sun*, como define o *Route RAcendeLuz* e quando o sensor desactiva desliga a luz como define o *Route RApagaLuz*. De seguida, o código para a resolução do problema.

```
curitiba::math::vec3 center(0,0,0), size(60,60,60);
curitiba::event_::Sensor *p=curitiba::event_::SensorFactory::create("ProximitySensor");
p->setProximitySensor("Prox",false,center,size);
Route *rAproxima= new Route("MainCamera","Prox","POSITION","PROXIMITY");
Route *rAcendeLuz= new Route("Prox","Sun","ENTER","LIGHT_ON");
Route *rApagaLuz= new Route("Prox","Sun","EXIT","LIGHT_OFF");
```

Exemplo 2: Suponhamos que pretendemos activar uma animação de posição de um objecto. No exemplo que se segue criamos um sensor de tempo que inicialmente está inactivo, um sensor de proximidade na origem com tamanho 60 nos três eixos e um interpolador de posição. Quando a câmara se aproxima do sensor de proximidade este

activa o sensor de tempo que faz com que o interpolador de posição anime o objecto. De seguida, apresento o código para a resolução do problema.

```
curitiba::math::vec3 center(0,0,0), size(60,60,60);
Sensor *ps= curitiba::event_::SensorFactory::create("ProximitySensor");
ps->setProximitySensor("Prox", false, center, size);
Sensor *ts= curitiba::event_::SensorFactory::create("TimeSensor");
ts->setTimeSensor("tsensor",true,0,10,true);
PositionInterpolator *pin= curitiba::event_::InterpolatorFactory:: create("PositionInterpolator");
vector<float> k;
k.push_back(0);
k.push_back(1);
vec3 c(RENDERMANAGER->getScene("MainScene")->getSceneObject("polySurface1219")-
>getBoundingVolume()->getCenter());
vec3 point1(c.x,c.y,c.z);
vec3 point2(c.x,c.y+1,c.z);
vector<vec3> kv;
kv.push_back(point1);
kv.push_back(point2);
pin->setPositionInterpolator("pinterpolator", k,kv);
ISceneObject *o=RENDERMANAGER->getScene(
    "MainScene")->getSceneObject("polySurface1219");
ObjectAnimation *obj= new ObjectAnimation(“Obj”, o);
Route *r0= new Route(“tsensor","pinterpolator",
    "TIMESENSOR_FRACTION_CHANGED", "INTERPOLATOR_FRACTION_CHANGED");
Route *r1= new Route(“pinterpolator”, “Obj”,
    “INTERPOLATOR_POSITION”, ”OBJECT_FRACTION”);
Route *r2= new Route("Prox","tsensor","ENTER","ACTIVATE");
Route *r3= new Route("Prox","tsensor","EXIT","DEACTIVATE");
```

No exemplo acima, é introduzida uma nova classe de objectos: *ObjectAnimation*. Esta classe permite realizar animações em objectos, neste caso uma translação, evitando assim restringir a classe dos interpoladores de posição à translação de objectos. Este objecto acede à transformação geométrica do objecto a animar e realize a translação pretendida, de acordo com os dados recebidos do interpolador.

3.6. Carregar cenas dinâmicas por XML

Como já foi referido, pretende-se neste projecto que o fluir dos eventos possa ser especificado nos ficheiros de projecto que permitem definir cenas, com recurso à notação XML.

Esta terminologia é mais simples para um utilizador não familiarizado com a programação por código. Por outro lado, não necessita de conhecer os métodos de

classe para a criação dos objectos, basta conhecer os atributos da classe e o Curitiba encarrega-se de interpretar o script e criar os procedimentos e objectos necessários ao seu funcionamento.

Sobretudo traz outras vantagens ao nível da facilidade e rapidez de experimentação de exemplos, pois o utilizador não tem necessidade de estar constantemente a recompilar o código de cada vez que se pretende experimentar diferentes tipos de interacção.

Portanto os sensores, interpoladores e *routes* podem ser carregados também no ficheiro de configuração em XML, assim como as restantes definições do Curitiba que já o permitiam. Desta forma foi alterado o *projectloader* do Curitiba para carregar estes objectos. Os seguintes exemplos mostram como podemos carregar os mesmos objectos dos exemplos anteriores, mas desta vez, usando as definições XML.

Exemplo 1:

```
<sensors>
        <sensor name="prox" class="ProximitySensor">
                <enabled value="0" />
                <center x="0" y="0" z="-62" />
                <size x="60" y="60" z="60" />
        </sensor>
</sensors>
<routes>
        <route sender="MainCamera" receiver="prox"
           eventIn="POSITION" eventOut="PROXIMITY">
        </route>
        <route sender="prox" receiver="Sun"
                eventIn="ENTER" eventOut="LIGHT_ON">
        </route>
        <route sender="prox" receiver="Sun"
                eventIn="EXIT" eventOut="LIGHT_OFF">
        </route>
</routes>
```

Exemplo 2:

```
<sensors>
        <sensor name="prox" class="ProximitySensor">
                <enabled value="0" />
                <center x="0" y="0" z="-62" />
                <size x="60" y="60" z="60" />
        </sensor>
        <sensor name="tsensor" class="TimeSensor">
                <enabled value="0" />
                <secondsToStart value="0" />
                <cicleInterval value="10" />
                <loop value="1" />
        </sensor>
</sensors>
<interpolators>
        <interpolator name="posInter" class="PositionInterpolator"
                 object="polySurface1219"
                 keys="0 1"
                 keyValues="0 0 -62, 0 1 -62">
        </interpolator>
</interpolators>
<routes>
        <route sender="MainCamera" receiver="prox"
                 eventIn="POSITION" eventOut="PROXIMITY">
      </route>
        <route sender="tsensor" receiver="posInter"
                  eventIn="TIMESENSOR_FRACTION_CHANGED"
                eventOut="INTERPOLATOR_FRACTION_CHANGED">
        </route>
        <route sender="prox" receiver="tsensor"
                eventIn="ENTER" eventOut="ACTIVATE">
        </route>
        <route sender="prox" receiver="tsensor"
                eventIn="EXIT" eventOut="DEACTIVATE">
        </route>
</routes>
```

Capítulo 4 - CONCLUSÕES E TRABALHO FUTURO

O objectivo traçado inicialmente para a tese foi atingido. Consistia em dotar o Curitiba da capacidade para definir e visualizar cenas dinâmicas e com interacção quer com o utilizador quer entre elementos da cena.

Para atingir esse objectivo foi criada uma arquitectura de classes genérica e facilmente expansível que faz a gestão dos eventos.

Esta arquitectura define e permite a expansão dos tipo de dados a circular pelo motor, dos eventos a enviar e a receber, dos sensores e interpoladores a implementar.

Pretendia-se que fosse possível desencadear uma sequência de eventos que despoletam e determinam o tempo de vida, se for caso disso, das animações e mudanças de estado. Desta forma, os objectos poderiam influenciar a execução uns dos outros, criando uma interacção mais rica e poderosa.

Foi criado um motor de eventos com grandes potencialidades, genérico e versátil. Permite assentar qualquer tipo de interacção, seja ela através de dispositivos de input ou entre objectos dentro de um ambiente.

A implementação de *routes* veio permitir definir de uma forma simples essa sequência de eventos, fazendo o encadeamento entre eles, permitindo mais interacção e mais abrangente.

A implementação de *routes*, usando o próprio motor de eventos, sem a necessidade de alterar absolutamente nada do núcleo do motor de eventos, foi a prova evidente de que o motor é versátil, genérico e expansível. Não só permite interacção entre o utilizador e os elementos da cena, mas também ilimitada interacção dos elementos entre si.

Com o reencaminhamento de eventos, usando o próprio motor, provou-se que é possível expandir mais o motor de eventos e o resultado dessa expansão potenciar ainda mais generalidade e versatilidade ao motor.

Foi também acrescentada a possibilidade de definir o fluir dos eventos nos ficheiros de projecto que permitem definir cenas, com recurso à notação XML. Ou seja, a definição por XML de cenas dinâmicas.

Esta terminologia é mais simples para um utilizador não familiarizado com a programação por código, mas também mais fácil e rápida para experimentar exemplos, pois não há necessidade de estar sempre a recompilar o código a cada alteração

introduzida. Desta forma, os sensores, interpoladores e *routes* podem ser carregados também no ficheiro de configuração em XML.

Relativamente ao trabalho futuro, poderá passar pela implementação dos *routes* no núcleo do motor de eventos, uma vez que este mecanismo é imprescindível para o funcionamento do motor de eventos, dentro dos objectivos definidos. Este procedimento traria também vantagens ao nível da eficiência de execução.

Poder-se-á também considerar que a criação de diferentes tipos de sensores, interpoladores e animação mais elaborada e complexa, como por exemplo, diferentes sistemas de partículas, animação de esqueleto, etc., será um ponto a completar, pois enriquecerá a interacção com o Curitiba, principalmente em termos de visualização gráfica dos efeitos da animação.

BIBLIOGRAFIA

Artigos

[Andreoli] Roberto Andreoli, Rosario De Chiara, Ugo Erra, Vittorio Scarano: *Interactive 3D Environments by using videogame engines, Ninth International Conference on Information Visualisation, University of Greenwich, London , UK, ieeexplore.ieee.org, 2005*

[Andujar] Carlos Andujar; Marta Fairén; Ferran Argelaguet: *A Cost-effective Approach for Developing Application-control GUIs for Virtual Environments, IEEE Symposium on 3D User Interfaces, Alexandria, Virginia USA, 2006*

[Bongart] Robert Bongart: *Efficient Simulation of Fluid Dynamics in a 3D Game Engine, Master of Science Thesis in Computer Science, Royal Institute of Technology, Sweden, 2007*

[Cavazza] Mark Cavazza, Simon Hartley, Jean-Luc Lugrin, Paolo Libardi and Mikael Le Bras: *New Behavioural Approaches for Virtual Environments, School of Computing, University of Teesside, England, with collaboration of Department of Electronics for Automation, University of Brescia, Italy, 2004*

[Deligiannidis] Leonidas Deligiannidis, William Urbanski: *Virtual Reality Interface to Conventional First-Person-View Shooting Computer Games, The University of Georgia, USA, Department of Computer Science, 2005*

[Gomes] Paulo César Rodacki Gomes, Vitor Fernando Pamplona: *M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API, Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital, SBGames2005, na Escola Politécnica da Universidade de São Paulo (USP), Brasil, 2005*

[Gross] Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S. Kc, Janak J. Parekh: *An Active Events Model for Systems Monitoring, Working conference on complex and dynamic systems, Brisbane, Australia, 2001*

[Gustavsson] Mikael Gustavsson: *3D Game Engine Design for Mobile Phones with OpenGL ES 2.0, Master's Thesis in Computer Science, Royal Institute of Technology, Sweden, 2008*

[Haque] Mohammed E. Haque, Pallab Dasgupta: *Architectural/Engineering Visualization using Game Engine, Proceedings of the 2008 ASEE Gulf-Southwest Annual Conference, The University of New Mexico, USA, 2008*

[Janc] Artur Janc, Matthew Jarmak, Steven Kolk, Robert W Martin, Owen Pedrotti: *Virtutopia: A Scalable Framework For Online Virtual Environments, Worcester Polytechnic Institute, Massachusetts, USA, 2007*

[Jiang] Hua Jiang, G. Drew Kessler, Jean Nonnemaker: *DEMIS: A Dynamic Event Model for Interactive Systems, Proceedings of the ACM symposium on Virtual reality software and technology, Hong Kong, China, 2002*

[Kim] S. Kim, D. Kwon, T. Park, B. Choi: *A simple event model in Java3D-based VRML browser, Computer Graphics & Geometry Internet-Journal, Moscow Engineering Physics Institute, Russia, 2002*

[Lester] James C Lester and William H Bares, Charles B Callaway, and Stuart G Towns: *Natural Language Generation Journeys to Interactive 3DWorlds, Workshop on Natural Language Generation, Ontario, Canada, 1998*

[Li] Tsai-Yen Li and Hung-Kai Ting: *An Intelligent User Interface with Motion Planning for 3D Navigation, Proceedings of the IEEE Virtual Reality, New Brunswick, New Jersey, USA, 2000*

[Lugrin] Jean-Luc Lugrin, Remi Chaignon, Marc Cavazza: *A High-level Event System for Augmented Reality, University of Teesside, England, 2006*

[Moreno] Aitor Moreno, Jan Heukamp, Jorge Posada, Alejandro García-Alonso: *VDesktop: Event Management and Physically Based Behaviour in Tabletop Displays, VICOMTech with colaboration of Euskal Herriko Unibertsitatea, Basque Coutry, 2007*

[Scoresby] Jon Scoresby, Brett E. Shelton, Tim Stowell, Chad Coats, Michael R. Capell: *Iterations of an Open-Source 3D Game Engine: Multiplayer Environments for Learners, Department of Instructional Technology and Learning Sciences, College of Education and Human Services, Utah State University, USA, 2007*

[Taniguchi] Masaaki Taniguchi: *Event Processing For Complicated Routes In VRML 2.0, Proceedings of the third symposium on Virtual reality, ACM, California, United States, 1998*

[Tulip] James Tulip, James Bekkema, Keith Nesbitt**:** *Multi-threaded Game Engine Design, Proceedings of the 3rd Australasian conference on Interactive entertainment, ACM, Perth, Australia, 2006*

[Wünsche] Burkhard C. Wünsche, Blazej Kot, Andrew Gits, Robert Amor, John Hosking and John Grundy: *A Framework for Game Engine Based Visualisations, Proceedings of Image and Vision Computing, New Zealand, 2005*

Sites

[Hannah] Object-Oriented Game Design. **In:** DevMaster.net. **Disponível em:** *http://www.devmaster.net/articles/oo-game-design/***. Acesso em:** *Abril de 2009.*

[lighthouse3D] VRML Interactive Tuturial. **In:** *LightHouse3D.* **Disponível em:** *http://www.lighthouse3d.com/vrml/tutorial/.* **Acesso em:** *Abril de 2009.*

[MotorDeFísica] Motor de física. **In:** *wikipédia: a enciclopédia livre*. **Disponível em:** *http://pt.wikipedia.org/wiki/Motor_de_f%C3%ADsica.* **Acesso em:** *Abril de 2009.*

[MotorDeJogo] Motor de jogo. **In:** *wikipédia: a enciclopédia livre*. **Disponível em:** *http://pt.wikipedia.org/wiki/Motor_de_jogo.* **Acesso em:** *Abril de 2009.*

[ODE] *OPEN DYNAMICS ENGINE.* **In:** *Open Dynamics Engine* **Disponível em:** *http://www.ode.org/ode-latest-userguide.html.* **Acesso em:** *Abril de 2009.*

[pl3D] Ponte de Lima 3D. **In:** Universidade do Minho. **Disponível em:** *www.di.uminho.pt/pl3d/index.html*. **Acesso em:** *Abril de 2009*.

[Pong] Pong. In**:** *wikipédia: a enciclopédia livre*. **Disponível em:** *http://pt.wikipedia.org/wiki/Pong.* **Acesso em:** *Abril de 2009*